Committer's Guide

The FreeBSD Documentation Project

$FreeBSD: doc/en_US.ISO8859-1/articles/committers-guide/article.sgml,v 1.311 2012/02/24 14:10:05 gjb Exp $

This document provides information for the FreeBSD committer community. All new committers should read this document before they start, and existing committers are strongly encouraged to review it from time to time.

Almost all FreeBSD developers have commit rights to one or more repositories. However, a few developers do not, and some of the information here applies to them as well. (For instance, some people only have rights to work with the Problem Report database). Please see Section 15 for more information.

This document may also be of interest to members of the FreeBSD community who want to learn more about how the project works.

Table of Contents

1 Administrative Details

Login Methods	ssh(1), protocol 2 only
Main Shell Host	`freefall.FreeBSD.org`
Main Subversion Root	`svn+ssh://svn.FreeBSD.org/base` (see also Section 3).
Main CVSROOT's	`dcvs.FreeBSD.org:/home/dcvs`, `pcvs.FreeBSD.org:/home/pcvs` (see also Section 3).
Internal Mailing Lists	developers (technically called all-developers), doc-developers, doc-committers, ports-developers, ports-committers, src-developers, src-committers. (Each project repository has its own -developers and -committers mailing lists. Archives for these lists may be found in files `/home/mail/`*`repository-name`*`-developers-archive` and `/home/mail/`*`repository-name`*`-committers-archive` on the `FreeBSD.org` cluster.)
Core Team monthly reports	`/home/core/public/monthly-reports` on the `FreeBSD.org` cluster.
Ports Management Team monthly reports	`/home/portmgr/public/monthly-reports` on the `FreeBSD.org` cluster.
Noteworthy SVN Branches	`stable/7` (7.X-STABLE), `stable/8` (8.X-STABLE), `head` (-CURRENT)

It is required that you use ssh(1) to connect to the project hosts. If you do not know anything about ssh(1), please see Section 10.

Useful links:

- FreeBSD Project Internal Pages (http://www.FreeBSD.org/internal/)
- FreeBSD Project Hosts (http://www.FreeBSD.org/internal/machines.html)
- FreeBSD Project Administrative Groups (http://www.FreeBSD.org/administration.html)

2 Commit Bit Types

The FreeBSD repository has a number of components which, when combined, support the basic operating system source, documentation, third party application ports infrastructure, and various maintained utilities. When FreeBSD commit bits are allocated, the areas of the tree where the bit may be used are specified. Generally, the areas associated with a bit reflect who authorized the allocation of the commit bit. Additional areas of authority may be added at a later date: when this occurs, the committer should follow normal commit bit allocation procedures for that area of the tree, seeking approval from the appropriate entity and possibly getting a mentor for that area for some period of time.

Committer Type	*Responsible*	*Tree Components*
src	core@	src/, doc/ subject to appropriate review
doc	doceng@	doc/, www/, src/ documentation
ports	portmgr@	ports/

Commit bits allocated prior to the development of the notion of areas of authority may be appropriate for use in many parts of the tree. However, common sense dictates that a committer who has not previously worked in an area of the tree seek review prior to committing, seek approval from the appropriate responsible party, and/or work with a mentor. Since the rules regarding code maintenance differ by area of the tree, this is as much for the benefit of the committer working in an area of less familiarity as it is for others working on the tree.

Committers are encouraged to seek review for their work as part of the normal development process, regardless of the area of the tree where the work is occurring.

2.1 Policy for `doc/` committer activity in `src/`

- doc committers may commit documentation changes to src files, such as man pages, READMEs, fortune databases, calendar files, and comment fixes without approval from a src committer, subject to the normal care and tending of commits.
- doc committers may commit minor src changes and fixes, such as build fixes, small features, etc, with an "Approved by" from a src committer.
- doc committers may seek an upgrade to a src commit bit by acquiring a mentor, who will propose the doc committer to core. When approved, they will be added to 'access' and the normal mentoring period will ensue, which will involve a continuing of “Approved by” for some period.
- "Approved by" is only acceptable from non-mentored src committers -- mentored committers can provide a "Reviewed by" but not an "Approved by".

3 Version Control System Operations

It is assumed that you are already familiar with the basic operation of the version control systems in use. Traditionally this was CVS, but as of June 2008, Subversion is used for the src tree.

The CVS Repository Meisters <`cvsadm@FreeBSD.org`> are the "owners" of the repository and are responsible for direct modification of it for the purposes of cleanup or fixing some unfortunate abuse of the version control system by a committer. Should you cause some repository accident, say a bad import or a bad tag creation, mail the responsible part of CVS Repository Meisters <`cvsadm@FreeBSD.org`>, as stated in the table below, (or call one of them) and report the problem. For very important issues affecting the entire tree—not just a specific area—you can contact the CVS Repository Meisters <`cvsadm@FreeBSD.org`>. Please do *not* contact the CVS Repository Meisters <`cvsadm@FreeBSD.org`> for repocopies or other things that the more specific teams can handle.

The only ones able to directly fiddle the repository bits on the repository hosts are the repomeisters. To enforce this, there are no login shells available on the repository machines, except to the repomeisters.

Note: Depending on the affected area of the repository, you should send your request for a repocopy to one of the following email addresses. Email sent to these addresses will be forwarded to the appropriate repomeisters.

- pcvs@ - regarding `/home/pcvs`, the ports repository
- dcvs@ - regarding `/home/dcvs`, the doc repository
- projcvs@ - regarding `/home/projcvs`, the third party projects repository

The FreeBSD repositories are currently split into four distinct parts, namely `doc`, `ports`, `projects` and `src`. These are combined under a single `CVSROOT` when distributed via **CVSup** for the convenience of our users. The `src` tree is automatically exported to CVS for compatibility reasons only (e.g. **CVSup**). The "official" `src` repository is not stored in **CVS** but in Subversion. The official and exported trees are not necessarily equal.

Note: Note that the sources for the FreeBSD website (http://www.FreeBSD.org) are contained within the `www` module of the `doc` repository.

The CVS repositories are hosted on the repository machines. Currently, each of the repositories above reside on the same physical machine, `ncvs.FreeBSD.org`, but to allow for the possibility of placing each on a separate machine in the future, there is a separate hostname for each that committers should use. Additionally, each repository is stored in a separate directory. The following table summarizes the situation.

Table 1. FreeBSD CVS Repositories, Hosts and Directories

Repository	Host	Directory
doc	dcvs.FreeBSD.org	/home/dcvs
ports	pcvs.FreeBSD.org	/home/pcvs
projects	projcvs.FreeBSD.org	/home/projcvs

CVS operations are done remotely by setting the `CVSROOT` environment variable to the appropriate host and top-level directory (for example, `dcvs.FreeBSD.org:/home/dcvs`), and doing the appropriate check-out/check-in operations. Many committers define aliases which expand to the correct **cvs** invocation for the appropriate repository.

For example, a tcsh(1) user may add the following to their `.cshrc` for this purpose:

```
alias dcvs cvs -d user@dcvs.FreeBSD.org:/home/dcvs
alias pcvs cvs -d user@pcvs.FreeBSD.org:/home/pcvs
alias projcvs cvs -d user@projcvs.FreeBSD.org:/home/projcvs
```

This way they can do all CVS operations locally and use *X*`cvs commit` for committing to the official CVS repository. Refer to the cvs(1) manual page for usage.

> **Note:** Please do *not* use `cvs checkout` or `update` with the official repository machine set as the CVS Root for keeping your source tree up to date. Remote CVS is not optimized for network distribution and requires a big work/administrative overhead on the server side. Please use our advanced `cvsup` distribution method for obtaining the repository bits, and only do the actual `commit` operation on the repository host. We provide an extensive cvsup replication network for this purpose, as well as give access to `cvsup-master` if you really need to stay current to the latest changes. `cvsup-master` has got the horsepower to deal with this, the repository master server does not. Jun Kuriyama <`kuriyama@FreeBSD.org`> is in charge of `cvsup-master`.

If you need to use CVS `add` and `delete` operations in a manner that is effectively a mv(1) operation, then a repository copy is in order rather than using CVS `add` and `delete`. In a repository copy, a repomeister will copy the file(s) to their new name and/or location and let you know when it is done. The purpose of a repository copy is to preserve file change history, or logs. We in the FreeBSD Project greatly value the change history that a version control system gives to the project.

CVS reference information, tutorials, and FAQs can be found at: http://www.cvshome.org/docs/. The information in Karl Fogel's chapters from "Open Source Development with CVS" (http://cvsbook.red-bean.com/cvsbook.html) is also very useful.

Dag-Erling C. Smørgrav <`des@FreeBSD.org`> also supplied the following "mini primer" for CVS.

1. Check out a module with the `co` or `checkout` command.

   ```
   % cvs checkout shazam
   ```

   This checks out a copy of the `shazam` module. If there is no `shazam` module in the modules file, it looks for a top-level directory named `shazam` instead.

   Table 2. Useful `cvs checkout` options

`-P`	Do not create empty directories
`-l`	Check out a single level, no subdirectories
`-r`*`rev`*	Check out revision, branch or tag *`rev`*
`-D`*`date`*	Check out the sources as they were on date *`date`*

   Practical FreeBSD examples:

   - Check out the `Tools` module, which corresponds to `ports/Tools`:

     ```
     % cvs co Tools
     ```

     You now have a directory named `ports/Tools` with subdirectories `portbuild`, `scripts`, and `CVS`.

   - Check out the same files, but with full path:

```
% cvs co ports/Tools
```

  You now have a directory named `ports`, with subdirectories `CVS` and `Tools`. The `ports/Tools` directory has subdirectories `CVS` and `scripts`, etc.

- Check out the directory `Tools`, but none of the subdirectories:

  ```
  % cvs co -l Tools
  ```

  You now have a directory named `Tools` with just one subdirectory named `CVS`.

- Check out the `Tools` module as it was when support for FreeBSD 5.X was dropped:

  ```
  % cvs co -rRELEASE_5_EOL Tools
  ```

  You will not be able to commit modifications, since `RELEASE_5_EOL` is a point in time, not a branch.

- Check out the `Tools` module as it was on March 25th, 2009:

  ```
  % cvs co -D'2009-03-25' Tools
  ```

  You will not be able to commit modifications.

- Check out the `Tools` module as it was one week ago:

  ```
  % cvs co -D'last week' Tools
  ```

  You will not be able to commit modifications.

Note that cvs stores metadata in subdirectories named `CVS`. Similarly, Subversion stores metadata in subdirectories named `.svn`.

Arguments to `-D` and `-r` are sticky, which means cvs will remember them later, e.g. when you do a `cvs update`.

2. Check the status of checked-out files with the `status` command.

   ```
   % cvs status shazam
   ```

   This displays the status of the file `shazam` or of every file in the `shazam` directory. For every file, the status is given as one of:

Up-to-date	File is up-to-date and unmodified.
Needs Patch	File is unmodified, but there is a newer revision in the repository.
Locally Modified	File is up-to-date, but modified.
Needs Merge	File is modified, and there is a newer revision in the repository.
File had conflicts on merge	There were conflicts the last time this file was updated, and they have not been resolved yet.

   You will also see the local revision and date, the revision number of the newest applicable version ("newest applicable" because if you have a sticky date, tag or branch, it may not be the actual newest revision), and any sticky tags, dates or options.

3. Once you have checked something out, you can update it with the `update` command.

   ```
   % cvs update shazam
   ```

   This updates the file `shazam` or the contents of the `shazam` directory to the latest version along the branch you checked out. If you checked out a "point in time", it does nothing unless the tags have moved in the repository or some other weird stuff is going on.

Useful options, in addition to those listed above for `checkout`:

`-d`	Check out any additional missing directories.
`-A`	Update to head of main branch.

If you checked out a module with `-r` or `-D`, running `cvs update` with a different `-r` or `-D` argument or with `-A` will select a new branch, revision or date. The `-A` option clears all sticky tags, dates or revisions whereas `-r` and `-D` set new ones.

Theoretically, specifying `HEAD` as the argument to `-r` will give you the same result as `-A`, but that is just theory.

The `-d` option is useful if:

- somebody has added subdirectories to the module you have checked out after you checked it out.
- you checked out with `-l`, and later change your mind and want to check out the subdirectories as well.
- you deleted some subdirectories and want to check them all back out.

Watch the output of the `cvs update` with care. The letter in front of each filename indicates what was done with it:

`U`	The file was updated without trouble.
`P`	The file was updated without trouble (you will only see this when working against a remote repository).
`M`	The file had been modified, and was merged without conflicts.
`C`	The file had been modified, and was merged with conflicts.

Merging is what happens if you check out a copy of some file, modify it, then someone else commits a change, and you run `cvs update`. CVS notices that you have made local changes, and tries to merge your changes with the changes between the version you originally checked out and the one you updated to. If the changes are to separate portions of the file, it will almost always work fine (though the result might not be syntactically or semantically correct).

CVS will print an `M` in front of every locally modified file even if there is no newer version in the repository, so `cvs update` is handy for getting a summary of what you have changed locally.

If you get a `C`, then your changes conflicted with the changes in the repository (the changes were to the same lines, or neighboring lines, or you changed the local file so much that `cvs` can not figure out how to apply the repository's changes). You will have to go through the file manually and resolve the conflicts; they will be marked with rows of <, = and > signs. For every conflict, there will be a marker line with seven < signs and the name of the file, followed by a chunk of what your local file contained, followed by a separator line with seven = signs, followed by the corresponding chunk in the repository version, followed by a marker line with seven > signs and the revision number you updated to.

4. View differences between the local version and the repository version with the `diff` command.

```
% cvs diff shazam
```

shows you every modification you have made to the `shazam` file or module.

Table 3. Useful `cvs diff` options

`-u`	Uses the unified diff format.
`-c`	Uses the context diff format.
`-N`	Shows missing or added files.

You always want to use `-u`, since unified diffs are much easier to read than almost any other diff format (in some circumstances, context diffs generated with the `-c` option may be better, but they are much bulkier). A unified diff consists of a series of hunks. Each hunk begins with a line that starts with two `@` signs and specifies where in the file the differences are and how many lines they span. This is followed by a number of lines; some (preceded by a blank) are context; some (preceded by a `-` sign) are outtakes and some (preceded by a `+`) are additions.

You can also diff against a different version than the one you checked out by specifying a version with `-r` or `-D` as in `checkout` or `update`, or even view the diffs between two arbitrary versions (without regard for what you have locally) by specifying *two* versions with `-r` or `-D`.

5. View log entries with the `log` command.

   ```
   % cvs log shazam
   ```

   If `shazam` is a file, this will print a *header* with information about this file, such as where in the repository this file is stored, which revision is the `HEAD` for this file, what branches this file is in, and any tags that are valid for this file. Then, for each revision of this file, a log message is printed. This includes the date and time of the commit, who did the commit, how many lines were added and/or deleted, and finally the log message that the committer who did the change wrote.

   If `shazam` is a directory, then the log information described above is printed for each file in the directory in turn. Unless you give the `-l` to `log`, the log for all subdirectories of `shazam` is printed too, in a recursive manner.

   Use the `log` command to view the history of one or more files, as it is stored in the CVS repository. You can even use it to view the log message of a specific revision, if you add the `-r`*`rev`* to the `log` command:

   ```
   % cvs log -r1.2 shazam
   ```

   This will print only the log message for revision `1.2` of file `shazam` if it is a file, or the log message for revision `1.2` of each file under `shazam` if it is a directory.

6. See who did what with the `annotate` command. This command shows you each line of the specified file or files, along with which user most recently changed that line.

   ```
   % cvs annotate shazam
   ```

7. Add new files with the `add` command.

   Create the file, `cvs add` it, then `cvs commit` it.

   Similarly, you can add new directories by creating them and then `cvs add`ing them. Note that you do not need to commit directories.

8. Remove obsolete files with the `remove` command.

   Remove the file, then `cvs rm` it, then `cvs commit` it.

9. Commit with the `commit` or `checkin` command.

   Table 4. Useful `cvs commit` options

`-f`	Force a commit of an unmodified file.

`-m`*`msg`*	Specify a commit message on the command line rather than invoking an editor.

The following are some Subversion examples related to the src repository. More (in-depth) information can be found at Subversion Primer (http://wiki.freebsd.org/SubversionPrimer) and List of things missing in Subversion when compared to CVS (http://wiki.freebsd.org/SubversionMissing). The notes at http://people.freebsd.org/~peter/svn_notes.txt might also be useful. Subversion is also described in-depth in Version Control with Subversion (http://svnbook-red-bean.com/).

- Check out the `head` branch:

  ```
  % svn co svn+ssh://svn.freebsd.org/base/head /usr/src
  ```

Good commit messages are important. They tell others why you did the changes you did, not just right here and now, but months or years from now when someone wonders why some seemingly illogical or inefficient piece of code sneaked into your source file. It is also an invaluable aid to deciding which changes to MFC and which not to MFC.

Commit messages should be clear, concise and provide a reasonable summary to give an indication of what was changed and why.

Commit messages should provide enough information to enable a third party to decide if the change is relevant to them and if they need to read the change itself.

Avoid committing several unrelated changes in one go. It makes merging difficult, and also makes it harder to determine which change is the culprit if a bug crops up.

Avoid committing style or whitespace fixes and functionality fixes in one go. It makes merging difficult, and also makes it harder to understand just what functional changes were made. In the case of documentation files, it can make the job of the translation teams more complicated, as it becomes difficult for them to determine exactly what content changes need to be translated.

Avoid committing changes to multiple files in one go with a generic, vague message. Instead, commit each file (or small, related groups of files) with tailored commit messages.

Before committing, *always*:

- verify which branch you are committing to, using `svn status`. This is only needed for the src tree, as the other trees are not branched.
- review your diffs, using the diff command of the version control system.

Also, ALWAYS specify which files to commit explicitly on the command line, so you do not accidentally commit other files than the ones you intended — a commit operation without any arguments usually will commit every modification in your current working directory and every subdirectory.

Additional tips and tricks:

1. You can place commonly used options in your `~/.cvsrc`, like this:

   ```
   cvs -z3
   diff -Nu
   update -Pd
   checkout -P
   ```

   This example says:

- always use compression level 3 when talking to a remote server. This is a life-saver when working over a slow connection.
- always use the `-N` (show added or removed files) and `-u` (unified diff format) options to diff(1).
- always use the `-P` (prune empty directories) and `-d` (check out new directories) options when updating.
- always use the `-P` (prune empty directories) option when checking out.

2. Use Eivind Eklund's `cdiff` script to view unidiffs. It is a wrapper for less(1) that adds ANSI color codes to make hunk headers, outtakes and additions stand out; context and garbage are unmodified. It also expands tabs properly (tabs often look wrong in diffs because of the extra character in front of each line).

   `textproc/cdiff`

   Simply use it instead of more(1) or less(1):

   ```
   % cvs diff -Nu shazam | cdiff
   ```

   Alternatively some editors like vim(1) (`editors/vim`) have color support and when used as a pager with color syntax highlighting switched on will highlight many types of file, including diffs, patches, and CVS/RCS logs.

   ```
   % echo "syn on" >> ~/.vimrc
   % cvs diff -Nu shazam | vim -
   % cvs log shazam | vim -
   ```

3. CVS is old, arcane, crufty and buggy, and sometimes exhibits non-deterministic behavior which some claim as proof that it is actually merely the Newtonian manifestation of a sentient transdimensional entity. It is not humanly possible to know its every quirk inside out, so do not be afraid to ask the resident AI (CVS Repository Meisters <`cvsadm@FreeBSD.org`>) for help.
4. Do not leave the `cvs commit` command in commit message editing mode for too long (more than 2–3 minutes). It locks the directory you are working with and will prevent other developers from committing into the same directory. If you have to type a long commit message, type it before executing `cvs commit` and insert it into the commit message or save it in a file before committing and use the `-F` option of CVS to read the commit message from that file, i.e.

   ```
   % vi logmsg
   % cvs ci -F logmsg shazam
   ```

   This is the fastest way of passing a commit message to CVS but you should be careful when editing the `logmsg` file before the commit, because CVS will not give you a chance to edit the message when you do the actual commit.
5. Speed up your CVS operation considerably by using a persistent ssh connection to the repository machine. First, put this configuration into your `~/.ssh/config`:

   ```
   Host dcvs.FreeBSD.org
         ControlPath /home/user/.ssh/cvs.cpath
   Host projcvs.FreeBSD.org
         ControlPath /home/user/.ssh/cvs.cpath
   Host pcvs.FreeBSD.org
         ControlPath /home/user/.ssh/cvs.cpath
   ```

   Now open the persistent connection to the repoman:

   ```
   % ssh -fNM ncvs.FreeBSD.org
   ```

The CVS commands should now respond faster, as they are reusing existing connection with the repository. Note that all the hostnames are case sensitive.

4 Subversion Primer

4.1 Introduction

The FreeBSD source repository switched from CVS to Subversion on May 31st, 2008. The first real SVN commit is *r179447*.

There are mechanisms in place to automatically merge changes back from the Subversion repository to the CVS one, so regular users should not notice a difference, however developers most certainly will.

Subversion is not that different from CVS when it comes to daily use, but there are differences. Subversion has a number of features that should make developers' lives easier. The most important advantage to Subversion (and the reason why FreeBSD switched) is that it handles branches and merging much better than CVS does. Some of the principal differences are:

- Commits are atomic.
- Revision numbers apply across the repository—all files that were modified in the same commit have the same revision number.
- Branching and tagging are namespace operations.
- Directories are versioned.
- Files and directories can have arbitrary, versioned metadata attached to them.
- Files and directories can be copied, with full history tracking.
- No more contortions due to CVS weakness such as applying patch(1) files at compile time in order to avoid touching vendor branch code.
- No more repo-copies.

Subversion can be installed from the FreeBSD Ports Collection, by issuing the following commands:

```
# cd /usr/ports/devel/subversion
# make clean install
```

4.2 Getting Started

There are three ways to obtain a working copy of the tree from Subversion. This section will explain them.

4.2.1 Direct Checkout

The first is to check out directly from the main repository:

```
% svn checkout svn+ssh://svn.freebsd.org/base/head /usr/src
```

The above command will check out a `CURRENT` source tree as */usr/src/*, which can be any target directory on the local filesystem. Omitting the final argument of that command causes the working copy, in this case, to be named “head”, but that can be renamed safely.

`svn+ssh` means the SVN protocol tunnelled over SSH. The name of the server is `svn.freebsd.org`, `base` is the path to the repository, and `head` is the subdirectory within the repository.

If your FreeBSD login name is different from your login name on your local machine, you must either include it in the URL (for example `svn+ssh://jarjar@svn.freebsd.org/base/head`), or add an entry to your `~/.ssh/config` in the form:

```
Host svn.freebsd.org
        User jarjar
```

This is the simplest method, but it's hard to tell just yet how much load it will place on the repository. Subversion is much faster than CVS, however.

> **Note:** The `svn diff` does not require access to the server as SVN stores a reference copy of every file in the working copy. This, however, means that Subversion working copies are very large in size.

4.2.2 Checkout from a Mirror

You can check out a working copy from a mirror by simply substituting the mirror's URL for `svn+ssh://svn.freebsd.org/base`. This can be an official mirror or a mirror you maintain yourself using `svnsync` or similar.

There is a serious disadvantage to this method: every time something is to be committed, a `svn switch --relocate` to the master repository has to be done, remembering to `svn switch` back to the mirror after the commit. Also, since `svn switch` only works between repositories that have the same UUID, some hacking of the local repository's UUID has to occur before it is possible to start using it.

Unlike with CVS and csup, the hassle of a local `svnsync` mirror probably is not worth it unless the network connectivity situation or other factors demand it. If it is needed, see the end of this chapter for information on how to set one up.

4.2.3 Checkout from a Local Mirror Using SVK

The third alternative is to use SVK to maintain a local mirror. It is a version control system build on top of Subversion's storage engine. It is identical to Subversion in most respects, except that it allows for setting up parts of repositories as mirrors of other repositories, and keeping local branches for merging back into the upstream repositories. There are extensions that allow SVK to mirror CVS and Perforce repositories in addition to Subversion ones.

Like everything, SVK has its disadvantages, one being that local revision numbers will not match upstream revision numbers. This makes it difficult to `svk log`, `svk diff`, or `svk update` to an arbitrary upstream revision.

To set up a mirror of the FreeBSD repository, do:

```
% svk mirror svn+ssh://svn.freebsd.org/base //freebsd/base
```

The local SVK repository will be stored in `~/.svk/local/`, but can be moved to whereever suits. If it is moved, `~/.svk/config` should be amended manually to reflect the move.

Any path can be used, not just the one in the example above. A common pattern is to place mirrors under `//mirror`, e.g. `//mirror/freebsd/base/`, and local branches under `//local`.

To pull down the contents of the repository to the mirror:

```
% svk sync //freebsd/base
```

> **Note:** `svk sync` will take a very long time, possibly several days over a slow network connection. Peter Wemm <`peter@FreeBSD.org`> has a tarball that can be used to jumpstart the mirror, but only if one does not exist already.

To use Peter Wemm <`peter@FreeBSD.org`>'s tarball mentioned in the note above:

```
% cd ~
% scp freefall:/home/peter/dot_svk_r179646.tbz2 .
% tar xf dot_svk_r179646.tbz2
```

Then edit `~/.svk/config` and replace `/scratch/tmp/peter/.svk/local/` with the equivalent of `/home/`*`jarjar`*`/.svk/local/`.

You can check out files directly from your mirror, once it has been created:

```
% svk checkout //freebsd/base/head /usr/src
```

Unlike SVN, SVK does not store metadata or reference copies in the working copy. All metadata is recorded in `~/.svk/config`; reference copies are not used at all because SVK always operates on a local repository.

When committing from a working copy like the one above, SVN will commit directly to the upstream repository, then syncronise the mirror.

However, the "killer app" for SVK is the ability to work without a network connection. To do that, a local branch must be set up:

```
% svk mkdir //local/freebsd
% svk copy //freebsd/base/head //local/freebsd/head
```

Once again, any path can be used, it does not have to specifically be the one in the example.

Before use, the local branch has to be synchronised, like so:

```
% svk pull //local/freebsd/head
```

Then check out from the newly created local branch:

```
% svk checkout //local/freebsd/head /usr/src
```

The point of this exercise is showing that it is possible to commit work-in-progress to a local branch, and only push it to the upstream repository when work is complete. The easy way to push is with `svk push`, but there is a serious disadvantage to it: it will push every single commit made to the local branch incrementally instead of lumping them all into a single commit. Therefore, using `svk smerge` is preferable.

4.2.4 `RELENG_*` Branches and General Layout

In `svn+ssh://svn.freebsd.org/base`, *base* refers to the source tree. Similarly, *ports* refers to the ports tree, and so on. These are separate repositories with their own change number sequences, access controls and commit mail.

For the base repository, HEAD refers to the -CURRENT tree. For example, `head/bin/ls` is what would go into `/usr/src/bin/ls` in a release. Some other key locations are:

- */stable/n* which corresponds to `RELENG_n`.
- */releng/n.n* which corresponds to `RELENG_n_n`.
- */release/n.n.n* which corresponds to `RELENG_n_n_n_RELEASE`.
- */vendor** is the vendor branch import work area. This directory itself does not contain branches, however its subdirectories do. This contrasts with the *stable*, *releng* and *release* directories.
- */projects* and */user* feature a branch work area, like in Perforce. As above, the */user* directory does not contain branches itself.

4.3 Daily Use

This section will explain how to perform common day-to-day operations with Subversion. There should be no difference between SVN and SVK in daily use, except for the revision renumbering mentioned earlier.

> **Note:** SVN and SVK commands that have direct CVS equivalents usually have the same name and abbreviations. For example: *checkout* and *co*, *update* and *up*, and *commit* and *ci*.

4.3.1 Help

Both SVN and SVK have built in help documentation. It can be accessed by typing the following command:

```
% svn help
```

4.3.2 Checkout

As seen earlier, to check out the FreeBSD head branch:

```
% svn checkout svn+ssh://svn.freebsd.org/base/head /usr/src
```

At some point, more than just `HEAD` will probably be useful, for instance when merging changes to stable/7. Therefore, it may be useful to have a partial checkout of the complete tree (a full checkout would be very painful).

To do this, first check out the root of the repository:

```
% svn checkout --depth=immediates svn+ssh://svn.freebsd.org/base
```

This will give `base` with all the files it contains (at the time of writing, just `ROADMAP.txt`) and empty subdirectories for `head`, `stable`, `vendor` and so on.

Expanding the working copy is possible. Just change the depth of the various subdirectories:

```
% svn up --set-depth=infinity base/head
% svn up --set-depth=immediates base/release base/releng base/stable
```

The above command will pull down a full copy of `head`, plus empty copies of every `release` tag, every `releng` branch, and every `stable` branch.

If at a later date merging to `7-STABLE` is required, expand the working copy:

```
% svn up --set-depth=infinity base/stable/7
```

Subtrees do not have to be expanded completely. For instance, expanding only `stable/7/sys` and then later expand the rest of `stable/7`:

```
% svn up --set-depth=infinity base/stable/7/sys
% svn up --set-depth=infinity base/stable/7
```

Updating the tree with `svn update` will only update what was previously asked for (in this case, `head` and `stable/7`; it will not pull down the whole tree.

It is useful to note that decreasing the depth of a working copy is not possible.

4.3.3 Anonymous Checkout

It is possible to anonymously check out the FreeBSD repository with Subversion. This will give access to a read-only tree that can be updated, but not committed to. To do this, use one of the following commands:

```
% svn co svn://svn.freebsd.org/base/head /usr/src
% svn co http://svn.freebsd.org/base/head /usr/src
```

4.3.4 Updating the Tree

To update a working copy to either the latest revision, or a specific revision:

```
% svn update
% svn update -r12345
```

4.3.5 Status

To view the local changes that have been made to the working copy:

```
% svn status
```

CVS has no direct equivalent of this command. The nearest would be `cvs up -N` which shows local changes and files that are out-of-date. Doing this in SVN is possible too, however:

```
% svn status --show-updates
```

4.3.6 Editing and Committing

Like CVS but unlike Perforce, SVN and SVK do not need to be told in advance about file editing.

`svn commit`works like the equivalent CVS command. To commit all changes in the current directory and all subdirectories:

```
% svn commit
```

To commit all changes in, for example, the *`lib/libfetch/`* and *`usr/bin/fetch/`* in a single operation:

```
% svn commit lib/libfetch usr/bin/fetch
```

4.3.7 Adding and Removing Files

> **Note:** Before adding files, get a copy of auto-props.txt (http://people.freebsd.org/~peter/auto-props.txt) and add it to `~/.subversion/config` according to the instructions in the file. If you added something before you've read this, you may use `svn rm --keep-local` for just added files, fix your config file and re-add them again. The initial config file is created when you first run a svn command, even something as simple as `svn help`.

As with CVS, files are added to a SVN repository with `svn add`. To add a file named *foo*, edit it, then:

```
% svn add foo
```

Files can be removed with `svn remove`:

```
% svn remove foo
```

Subversion does not require `rm`ing the file before `svn rm`ing it, and indeed complains if that happens.

It is possible to add directories with `svn add`:

```
% mkdir bar
% svn add bar
```

Although `svn mkdir` makes this easier by combining the creation of the directory and the adding of it:

```
% svn mkdir bar
```

In CVS, the directory is immediately created in the repository when you `cvs add` it; this is not the case in Subversion. Furthermore, unlike CVS, Subversion allows directories to be removed using `svn rm`, however there is no `svn rmdir`:

```
% svn rm bar
```

4.3.8 Copying and Moving Files

The following (obviously) creates a copy of `foo.c`, named `bar.c`:

```
% svn copy foo.c bar.c
```

To move and rename a file:

```
% svn move foo.c bar.c
```

The above command is the exact equivalent of:

```
% svn copy foo.c bar.c
% svn remove foo.c
```

Neither of these operations have equivalents in CVS.

4.3.9 Log and Annotate

`svn log` will show all the revisions that affect a directory and files within that directory in reverse chronological order, if run on a directory. This contrasts with `cvs log` in that CVS shows the complete log for each file in the directory, including duplicate entries for revisions that affect multiple files.

`svn annotate`, or equally `svn praise` or `svn blame`, is equivalent to `cvs annotate` in everything but output format.

4.3.10 Diffs

The `svn diff` displays changes to the working copy of the repository. SVN's diffs are unified by default, unlike CVS's, and SVN's include new files by default in the diff output.

Like `cvs diff`, `svn diff` can show the changes between two revisions of the same file:

```
% svn diff -r179453:179454 ROADMAP.txt
```

It can also show all changes for a specific changeset. The following will show what changes were made to the current directory and all subdirectories in changeset 179454:

```
% svn diff -c179454 .
```

4.3.11 Reverting

Local changes (including additions and deletions) can be reverted using `svn revert`. Unlike `cvs up -C`, it does not update out-of-date files—it just replaces them with pristine copies of the original version.

4.3.12 Conflicts

If a `svn update` resulted in a merge conflict, Subversion will remember which files have conflicts and refuse to commit any changes to those files until explicitly told that the conflicts have been resolved. The simple, not yet deprecated procedure is the following:

```
% svn resolved foo
```

However, the preferred procedure is:

```
% svn resolve --accept=working foo
```

The two examples are equivalent. Possible values for `--accept` are:

- `working`: use the version in your working directory (which one presumes has been edited to resolve the conflicts).
- `base`: use a pristine copy of the version you had before `svn update`, discarding your own changes, the conflicting changes, and possibly other intervening changes as well.
- `mine-full`: use what you had before `svn update`, including your own changes, but discarding the conflicting changes, and possibly other intervening changes as well.
- `theirs-full`: use the version that was retrieved when you did `svn update`, discarding your own changes.

4.4 Advanced Use

4.4.1 Sparse Checkouts

The equivalent to `cvs checkout -l`, which checks out a directory without its subdirectories, is `svn checkout -N`. Unlike CVS, SVN remembers the `-N` so that a `svn update` does not end up pulling down the subdirectories. In Subversion 1.5 and newer, `-N` has been deprecated in favour of the `--depth` option which allows for precise control. Therefore:

```
% svn checkout -N svn+ssh://svn.freebsd.org/base ~/freebsd
```

is equivalent to:

```
% svn checkout --depth=empty svn+ssh://svn.freebsd.org/base ~/freebsd
```

Valid arguments to `--depth` are:

- `empty`: the directory itself without any of its contents.
- `files`: the directory and any files it contains.
- `immediates`: the directory and any files and directories it contains, but none of the subdirectories' contents.
- `infinity`: anything.

The `--depth` option applies to many other commands, including `svn commit`, `svn revert`, and `svn diff`.

Since `--depth` is sticky, there is a `--set-depth` option for `svn update` that will change the selected depth. Thus, given the working copy produced by the previous example:

```
% cd ~/freebsd
% svn update --set-depth=immediates .
```

The above command will populate the working copy in *~/freebsd* with `ROADMAP.txt` and empty subdirectories, and nothing will happen when `svn update` is executed on the subdirectories. However, the following command will set the depth for head (in this case) to infinity, and fully populate it:

```
% svn update --set-depth=infinity head
```

4.4.2 Direct Operation

Certain operations can be performed directly on the repository, without touching the working copy. Specifically, this applies to any operation that does not require editing a file, including:

- `log`, `diff`.
- `mkdir`.
- `remove`, `copy`, `rename`.
- `propset`, `propedit`, `propdel`.
- `merge`.

Branching is very fast. The following command would be used to branch `RELENG_8`:

```
% svn copy svn+ssh://svn.freebsd.org/base/head svn+ssh://svn.freebsd.org/base/stable/8
```

This is equivalent to the following set of commands which take minutes and hours as opposed to seconds, depending on your network connection:

```
% svn checkout --depth=immediates svn+ssh://svn.freebsd.org/base
% cd base
% svn update --depth=infinity head
% svn copy head stable/8
% svn commit stable/8
```

4.4.3 Merging with SVN

This section deals with merging code from one branch to another (typically, from head to a stable branch). For information about vendor imports, see the next section in this primer.

> **Note:** In all examples below, `$FSVN` refers to the location of the FreeBSD Subversion repository, `svn+ssh://svn.freebsd.org/base/`.

4.4.3.1 About Merge Tracking

From the user's perspective, merge tracking information (or mergeinfo) is stored in a property called `svn:mergeinfo`, which is a comma-separated list of revisions and ranges of revisions that have been merged. When set on a file, it applies only to that file. When set on a directory, it applies to that directory and its descendants (files and directories) except for those that have their own `svn:mergeinfo`.

It is *not* inherited. For instance, `stable/6/contrib/openpam/` does not implicitly inherit mergeinfo from `stable/6/`, or `stable/6/contrib/`. Doing so would make partial checkouts very hard to manage. Instead, mergeinfo is explicitly propagated down the tree. For merging something into `branch/foo/bar/`, the following rules apply:

1. If `branch/foo/bar/` doesn't already have a mergeinfo record, but a direct ancestor (for instance, `branch/foo/`) does, then that record will be propagated down to `branch/foo/bar/` before information about the current merge is recorded.

2. Information about the current merge will *not* be propagated back up that ancestor.
3. If a direct descendant of `branch/foo/bar/` (for instance, `branch/foo/bar/baz/`) already has a mergeinfo record, information about the current merge will be propagated down to it.

If you consider the case where a revision changes several separate parts of the tree (for example, `branch/foo/bar/` and `branch/foo/quux/`), but you only want to merge some of it (for example, `branch/foo/bar/`), you will see that these rules make sense. If mergeinfo was propagated up, it would seem like that revision had also been merged to `branch/foo/quux/`, when in fact it had not been.

4.4.3.2 Selecting the Source and Target

Because of mergeinfo propagation, it is important to choose the source and target for the merge carefully to minimise property changes on unrelated directories.

The rules for selecting the merge target (the directory that you will merge the changes to) can be summarised as follows:

1. Never merge directly to a file.
2. Never, ever merge directly to a file.
3. *Never, ever, ever* merge directly to a file.
4. Changes to kernel code should be merged to `sys/`. For instance, a change to the ichwd(4) driver should be merged to `sys/`, not `sys/dev/ichwd/`. Likewise, a change to the TCP/IP stack should be merged to `sys/`, not `sys/netinet/`.
5. Changes to code under `etc/` should be merged at `etc/`, not below it.
6. Changes to vendor code (code in `contrib/`, `crypto/` and so on) should be merged to the directory where vendor imports happen. For instance, a change to `crypto/openssl/util/` should be merged to `crypto/openssl/`. This is rarely an issue, however, since changes to vendor code are usually merged wholesale.
7. Changes to userland programs should as a general rule be merged to the directory that contains the Makefile for that program. For instance, a change to `usr.bin/xlint/arch/i386/` should be merged to `usr.bin/xlint/`.
8. Changes to userland libraries should as a general rule be merged to the directory that contains the Makefile for that library. For instance, a change to `lib/libc/gen/` should be merged to `lib/libc/`.
9. There may be cases where it makes sense to deviate from the rules for userland programs and libraries. For instance, everything under `lib/libpam/` is merged to `lib/libpam/`, even though the library itself and all of the modules each have their own Makefile.
10. Changes to man pages should be merged to `share/man/man`*`N`*`/`, for the appropriate value of `N`.
11. Changes to a top-level file in the source tree such as `UPDATING` or `Makefile.inc1` should be merged directly to that file rather than to the root of the whole tree. Yes, this is an exception to the first three rules.
12. When in doubt, ask.

If you need to merge changes to several places at once (for instance, changing a kernel interface and every userland program that uses it), merge each target separately, then commit them together. For instance, if you merge a revision that changed a kernel API and updated all the userland bits that used that API, you would merge the kernel change to sys, and the userland bits to the appropriate userland directories, then commit all of these in one go.

The source will almost invariably be the same as the target. For instance, you will always merge `stable/7/lib/libc/` from `head/lib/libc/`. The only exception would be when merging changes to code that has moved in the source branch but not in the parent branch. For instance, a change to pkill(1) would be merged from `bin/pkill/` in head to `usr.bin/pkill/` in stable/7.

4.4.3.3 Preparing the Merge Target

Because of the mergeinfo propagation issues described earlier, it is very important that you never merge changes into a sparse working copy. You must always have a full checkout of the branch you will merge into. For instance, when merging from HEAD to 7, you must have a full checkout of stable/7:

```
% cd stable/7
% svn up --set-depth=infinity
```

The target directory must also be up-to-date and must not contain any uncommitted changes or stray files.

4.4.3.4 Identifying Revisions

Identifying revisions to be merged is a must. If the target already has complete mergeinfo, ask SVN for a list:

```
% cd stable/6/contrib/openpam
% svn mergeinfo --show-revs=eligible $FSVN/head/contrib/openpam
```

If the target does not have complete mergeinfo, check the log for the merge source.

4.4.3.5 Merging

Now, let's start merging!

4.4.3.5.1 The Principles

Say you would like to merge:

- revision `$R`.
- in directory $target in stable branch $B.
- from directory $source in head.
- $FSVN is `svn+ssh://svn.freebsd.org/base`.

Assuming that revisions $P and $Q have already been merged, and that the current directory is an up-to-date working copy of stable/$B, the existing mergeinfo looks like this:

```
% svn propget svn:mergeinfo -R $target
$target - /head/$source:$P,$Q
```

Merging is done like so:

```
% svn merge -c$R $FSVN/head/$source $target
```

Checking the results of this is possible with `svn diff`.

The svn:mergeinfo now looks like:

```
% svn propget svn:mergeinfo -R $target
$target - head/$source:$P,$Q,$R
```

If the results are not exactly as shown, assistance may be required before committing as mistakes may have been made, or there may be something wrong with the existing mergeinfo, or there may be a bug in Subversion.

4.4.3.5.2 Merging into the Kernel (`sys/`)

As stated above, merging into the kernel is different from merging in the rest of the tree. In many ways merging to the kernel is simpler because there is always the same merge target (`sys/`).

Once `svn merge` has been executed, `svn diff` has to be run on the directory to check the changes. This may show some unrelated property changes, but these can be ignored. Next, build and test the kernel, and, once the tests are complete, commit the code as normal, making sure that the commit message starts with "Merge *`r226222`* from head", or similar.

4.4.3.6 Precautions Before Committing

As always, build world (or appropriate parts of it).

Check the changes with `svn diff` and `svn stat`. Make sure all the files that should have been added or deleted were in fact added or deleted.

Take a closer look at any property change (marked by a `M` in the second column of `svn stat`). Normally, no svn:mergeinfo properties should be anywhere except the target directory (or directories).

If something looks fishy, ask for help.

4.4.3.7 Committing

Make sure to commit a top level directory to have the mergeinfo included as well. Do not specify individual files on the command line. For more information about committing files in general, see the relevant section of this primer.

4.4.4 Reverting a Commit

Reverting a commit to a previous version is fairly easy:

```
% svn merge -r179454:179453 ROADMAP.txt
% svn commit
```

Change number syntax, with negative meaning a reverse change, can also be used:

```
% svn merge -c -179454 ROADMAP.txt
% svn commit
```

This can also be done directly in the repository:

```
% svn merge -r179454:179453 svn+ssh://svn.freebsd.org/base/ROADMAP.txt
```

Reverting the deletion of a file is slightly different. Copying the version of the file that predates the deletion is required. For example, to restore a file that was deleted in revision N, restore version N-1:

```
% svn copy svn+ssh://svn.freebsd.org/base/ROADMAP.txt@179454
% svn commit
```

or, equally:

```
% svn copy svn+ssh://svn.freebsd.org/base/ROADMAP.txt@179454 svn+ssh://svn.freebsd.org/base
```

Do *not* simply recreate the file manually and `svn add` it—this will cause history to be lost.

4.4.5 Fixing Mistakes

While we can do surgery in an emergency, do not plan on having mistakes fixed behind the scenes. Plan on mistakes remaining in the logs forever. Be sure to check the output of `svn status` and `svn diff` before committing.

Mistakes will happen, but, unlike with CVS, they can generally be fixed without disruption.

Take a case of adding a file in the wrong location. The right thing to do is to `svn move` the file to the correct location and commit. This causes just a couple of lines of metadata in the repository journal, and the logs are all linked up correctly.

The wrong thing to do is to delete the file and then `svn add` an independent copy in the correct location. Instead of a couple of lines of text, the repository journal grows an entire new copy of the file. This is a waste.

4.4.6 Setting up a svnsync Mirror

You probably do not want to do this unless there is a good reason for it. Such reasons might be to support many multiple local read-only client machines, or if your network bandwidth is limited. Starting a fresh mirror from empty would take a very long time. Expect a minimum of 10 hours for high speed connectivity. If you have international links, expect this to take 4 to 10 times longer.

A far better option is to grab a seed file. It is large (~1GB) but will consume less network traffic and take less time to fetch than a svnsync will. This is possible in one of the following three ways:

```
% rsync -va --partial --progress freefall:/home/peter/svnmirror-base-r179637.tbz2 .
```

```
% rsync -va --partial --progress rsync://repoman.freebsd.org:50873/svnseed/svnmirror-base-r215629.tar.xz .
```

```
% fetch ftp://ftp.freebsd.org/pub/FreeBSD/development/subversion/svnmirror-base-r221445.tar.xz
```

Once you have the file, extract it to somewhere like `home/svnmirror/base/`. Then, update it, so that it fetches changes since the last revision in the archive:

```
% svnsync sync file:///home/svnmirror/base
```

You can then set that up to run from cron(8), do checkouts locally, set up a svnserve server for your local machines to talk to, etc.

The seed mirror is set to fetch from `svn://svn.freebsd.org/base`. The configuration for the mirror is stored in `revprop 0` on the local mirror. To see the configuration, try:

```
% svn proplist -v --revprop -r 0 file:///home/svnmirror/base
```

Use `propset` to change things.

4.4.7 Committing High-ASCII Data

Files that have high-ASCII bits are considered binary files in SVN, so the pre-commit checks fail and indicate that the `mime-type` property should be set to `application/octet-stream`. However, the use of this is discouraged, so please do not set it. The best way is always avoiding high-ASCII data, so that it can be read everywhere with any text editor but if it is not avoidable, instead of changing the mime-type, set the `fbsd:notbinary` property with `propset`:

```
% svn propset fbsd:notbinary yes foo.data
```

4.4.8 Maintaining a Project Branch

A project branch is one that's synced to head (or another branch) is used to develop a project then commit it back to head. In SVN, "dolphin" branching is used for this. A "dolphin" branch is one that diverges for a while and is finally committed back to the original branch. During development code migration in one direction (from head to the branch only). No code is committed back to head until the end. Once you commit back at the end, the branch is dead (although you can have a new branch with the same name after you delete the branch if you want).

As per http://people.freebsd.org/~peter/svn_notes.txt, work that is intended to be merged back into HEAD should be in `base/projects/`. If you are doing work that is beneficial to the FreeBSD community in some way but not intended to be merged directly back into HEAD then the proper location is `base/user/`*`your-name`*`/`. This page (http://svnweb.freebsd.org/base/projects/GUIDELINES.txt) contains further details.

To create a project branch:

```
% svn copy svn+ssh://svn.freebsd.org/base/head svn+ssh://svn.freebsd.org/base/projects/spif
```

To merge changes from HEAD back into the project branch:

```
% cd copy_of_spif
% svn merge svn+ssh://svn.freebsd.org/base/head
% svn commit
```

It is important to resolve any merge conflicts before committing.

To collapse everything back at the end:

```
% svn write me
```

4.5 Some Tips

In commit logs etc., "rev 179872" should be spelled "r179872" as per convention.

Speeding up checkouts and minimising network traffic is possible with the following recipe:

```
% svn co --depth=empty svn+ssh://svn.freebsd.org/base fbsvn
% cd fbsvn
% svn up --depth=empty stable
% svn up head
% cd stable
% cp -r ../head/ 7
% cd 7
% svn switch svn+ssh://svn.freebsd.org/base/stable/7
% cd ..
% cp -r 7/ 6
% cd 6
% svn switch svn+ssh://svn.freebsd.org/base/stable/6
```

What this bit of evil does is check out head, stable/7 and stable/6. We create the empty checkout directories under SVN's control. In SVN, subtrees are self identifying, like in CVS. We check out head and clone it as stable/7. Except we don't want the head version so we "switch" it to the 7.x tree location. SVN downloads diffs to convert the "head" files to "stable/7" instead of doing a fresh checkout. The same goes for stable/6. This does, however, definitely count as abuse of the working copy client code!

Checking out a working copy with a stock Subversion client withouth FreeBSD-specific patches (`WITH_FREEBSD_TEMPLATE`) will mean that `$FreeBSD$` tags will not be expanded. Once the correct version has been installed, trick Subversion into expanding them like so:

```
% svn propdel -R svn:keywords .
% svn revert -R .
```

This is not a good idea if uncommitted patches exist, however.

5 Conventions and Traditions

As a new developer there are a number of things you should do first. The first set is specific to committers only. (If you are not a committer, e.g. have GNATS-only access, then your mentor needs to do these things for you.)

5.1 Guidelines For Committers

If you have been given commit rights to one or more of the repositories:

- Add your author entity to `doc/en_US.ISO8859-1/share/sgml/authors.ent`; this should be done first since an omission of this commit will cause the next commits to break the doc/ build.

  This is a relatively easy task, but remains a good first test of your CVS skills.

  Note: Don't forget to get mentor approval for these patches!

- Also add your author entity to `www/en/developers.sgml`.
- Add yourself to the "Developers" section of the Contributors List (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/contributors/index.html) (`doc/en_US.ISO8859-1/articles/contributors/contrib.committers.sgml`) and remove yourself from the "Additional Contributors" section

(`doc/en_US.ISO8859-1/articles/contributors/contrib.additional.sgml`). Please note that entries are sorted by last name.

- Add an entry for yourself to `www/share/sgml/news.xml`. Look for the other entries that look like "A new committer" and follow the format.
- You should add your PGP or GnuPG key to `doc/share/pgpkeys` (and if you do not have a key, you should create one). Do not forget to commit the updated `doc/share/pgpkeys/pgpkeys.ent` and `doc/share/pgpkeys/pgpkeys-developers.sgml`. Please note that entries are sorted by last name.

  Dag-Erling C. Smørgrav <`des@FreeBSD.org`> has written a shell script (`doc/share/pgpkeys/addkey.sh`) to make this extremely simple. See the README (http://cvsweb.FreeBSD.org/doc/share/pgpkeys/README) file for more information.

  > **Note:** It is important to have an up-to-date PGP/GnuPG key in the Handbook, since the key may be required for positive identification of a committer, e.g. by the FreeBSD Administrators <`admins@FreeBSD.org`> for account recovery. A complete keyring of `FreeBSD.org` users is available for download from http://www.FreeBSD.org/doc/pgpkeyring.txt.

- Add an entry for yourself to `src/share/misc/committers-`*`repository`*`.dot`, where repository is either doc, ports or src, depending on the commit privileges you obtained.
- Some people add an entry for themselves to `ports/astro/xearth/files/freebsd.committers.markers`.
- Some people add an entry for themselves to `src/usr.bin/calendar/calendars/calendar.freebsd`.
- If you already have an account at the FreeBSD wiki (http://wiki.freebsd.org), make sure your mentor moves you from the Contributors group (http://wiki.freebsd.org/ContributorsGroup) to the Developers group (http://wiki.freebsd.org/DevelopersGroup). Otherwise, consider signing up for an account so you can publish projects and ideas you are working on.
- (For committers only:) If you subscribe to svn-src-all (http://lists.FreeBSD.org/mailman/listinfo/svn-src-all) or the FreeBSD CVS commit message mailing list (http://lists.FreeBSD.org/mailman/listinfo/cvs-all), you will probably want to unsubscribe to avoid receiving duplicate copies of commit messages and their followups.

> **Note:** All `src` commits should go to FreeBSD-CURRENT first before being merged to FreeBSD-STABLE. No major new features or high-risk modifications should be made to the FreeBSD-STABLE branch.

5.2 Guidelines For Everyone

Whether or not you have commit rights:

- Introduce yourself to the other developers, otherwise no one will have any idea who you are or what you are working on. You do not have to write a comprehensive biography, just write a paragraph or two about who you are and what you plan to be working on as a developer in FreeBSD. (You should also mention who your mentor will be). Email this to the FreeBSD developers mailing list and you will be on your way!

- Log into `hub.FreeBSD.org` and create a `/var/forward/user` (where `user` is your username) file containing the e-mail address where you want mail addressed to *yourusername*@FreeBSD.org to be forwarded. This includes all of the commit messages as well as any other mail addressed to the FreeBSD committer's mailing list and the FreeBSD developers mailing list. Really large mailboxes which have taken up permanent residence on `hub` often get "accidentally" truncated without warning, so forward it or read it and you will not lose it.

  Due to the severe load dealing with SPAM places on the central mail servers that do the mailing list processing the front-end server does do some basic checks and will drop some messages based on these checks. At the moment proper DNS information for the connecting host is the only check in place but that may change. Some people blame these checks for bouncing valid email. If you want these checks turned off for your email you can place a file named `.spam_lover` in your home directory on `freefall.FreeBSD.org` to disable the checks for your email.

  Note: If you are a developer but not a committer, you will not be subscribed to the committers or developers mailing lists; the subscriptions are derived from the access rights.

5.3 Mentors

All new developers also have a mentor assigned to them for the first few months. Your mentor is responsible for teaching you the rules and conventions of the project and guiding your first steps in the developer community. Your mentor is also personally responsible for your actions during this initial period.

For committers: until your mentor decides (and announces with a forced commit to `access`) that you have learned the ropes and are ready to commit on your own, you should not commit anything without first getting your mentor's review and approval, and you should document that approval with an `Approved by:` line in the commit message.

6 Preferred License for New Files

Currently the FreeBSD Project suggests and uses the following text as the preferred license scheme:

```
/*-
 * Copyright (c) [year] [your name]
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 * [id for your version control system, if any]
 */
```

The FreeBSD project strongly discourages the so-called "advertising clause" in new code. Due to the large number of contributors to the FreeBSD project, complying with this clause for many commercial vendors has become difficult. If you have code in the tree with the advertising clause, please consider removing it. In fact, please consider using the above license for your code.

The FreeBSD project discourages completely new licenses and variations on the standard licenses. New licenses require the approval of <`core@FreeBSD.org`> to reside in the main repository. The more different licenses that are used in the tree, the more problems that this causes to those wishing to utilize this code, typically from unintended consequences from a poorly worded license.

Project policy dictates that code under some non-BSD licenses must be placed only in specific sections of the repository, and in some cases, compilation must be conditional or even disabled by default. For example, the GENERIC kernel must be compiled under only licenses identical to or substantially similar to the BSD license. GPL, APSL, CDDL, etc, licensed software must not be compiled into GENERIC.

Developers are reminded that in open source, getting "open" right is just as important as getting "source" right, as improper handling of intellectual property has serious consequences. Any questions or concerns should immediately be brought to the attention of the core team.

7 Developer Relations

If you are working directly on your own code or on code which is already well established as your responsibility, then there is probably little need to check with other committers before jumping in with a commit. If you see a bug in an area of the system which is clearly orphaned (and there are a few such areas, to our shame), the same applies. If, however, you are about to modify something which is clearly being actively maintained by someone else (and it is only by watching the *`repository`*`-committers` mailing list that you can really get a feel for just what is and is not) then consider sending the change to them instead, just as you would have before becoming a committer. For ports, you should contact the listed `MAINTAINER` in the `Makefile`. For other parts of the repository, if you are unsure who the active maintainer might be, it may help to scan the revision history to see who has committed changes in the past. Bill Fenner <`fenner@FreeBSD.org`> has written a nice shell script that can help determine who the active maintainer might be. It lists each person who has committed to a given file along with the number of commits each person has made. It can be found on `freefall` at `~fenner/bin/whodid`. If your queries go unanswered or the committer otherwise indicates a lack of interest in the area affected, go ahead and commit it.

If you are unsure about a commit for any reason at all, have it reviewed by `-hackers` before committing. Better to have it flamed then and there rather than when it is part of the repository. If you do happen to commit something which results in controversy erupting, you may also wish to consider backing the change out again until the matter is settled. Remember – with a version control system we can always change it back.

Do not impugn the intentions of someone you disagree with. If they see a different solution to a problem than you, or even a different problem, it is not because they are stupid, because they have questionable parentage, or because they are trying to destroy your hard work, personal image, or FreeBSD, but simply because they have a different outlook on the world. Different is good.

Disagree honestly. Argue your position from its merits, be honest about any shortcomings it may have, and be open to seeing their solution, or even their vision of the problem, with an open mind.

Accept correction. We are all fallible. When you have made a mistake, apologize and get on with life. Do not beat up yourself, and certainly do not beat up others for your mistake. Do not waste time on embarrassment or recrimination, just fix the problem and move on.

Ask for help. Seek out (and give) peer reviews. One of the ways open source software is supposed to excel is in the number of eyeballs applied to it; this does not apply if nobody will review code.

8 GNATS

The FreeBSD Project utilizes **GNATS** for tracking bugs and change requests. Be sure that if you commit a fix or suggestion found in a **GNATS** PR, you use `edit-pr` *`pr-number`* on `freefall` to close it. It is also considered nice if you take time to close any PRs associated with your commits, if appropriate. You can also make use of send-pr(1) yourself for proposing any change which you feel should probably be made, pending a more extensive peer-review first.

You can find out more about **GNATS** at:

- FreeBSD Problem Report Handling Guidelines (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/pr-guidelines/index.html)
- http://www.cs.utah.edu/csinfo/texinfo/gnats/gnats.html
- http://www.FreeBSD.org/support.html
- send-pr(1)

You can run a local copy of GNATS, and then integrate the FreeBSD GNATS tree in to it using CVSup. Then you can run GNATS commands locally. This lets you query the PR database without needing to be connected to the Internet.

Using a local GNATS tree

1. If you are not already downloading the GNATS tree, add this line to your `supfile`, and re-sup. Note that since GNATS is not under CVS control it has no tag, so if you are adding it to your existing `supfile` it should appear before any "tag=" entry as these remain active once set.

   ```
   gnats release=current prefix=/usr
   ```

   This will place the FreeBSD GNATS tree in `/usr/gnats`. You can use a *refuse* file to control which categories to receive. For example, to only receive `docs` PRs, put this line in `/usr/local/etc/cvsup/sup/refuse`[1].

   ```
   gnats/[a-ce-z]*
   ```

   The rest of these examples assume you have only supped the `docs` category. Adjust them as necessary, depending on the categories you are syncing.

2. Install the GNATS port from `ports/databases/gnats`. This will place the various GNATS directories under `$PREFIX/share/gnats`.

3. Symlink the GNATS directories you are supping under the version of GNATS you have installed.

   ```
   # cd /usr/local/share/gnats/gnats-db
   # ln -s /usr/gnats/docs
   ```

   Repeat as necessary, depending on how many GNATS categories you are syncing.

4. Update the GNATS `categories` file with these categories. The file is `$PREFIX/share/gnats/gnats-db/gnats-adm/categories`.

   ```
   # This category is mandatory
   pending:Category for faulty PRs:gnats-admin:
   #
   # FreeBSD categories
   #
   docs:Documentation Bug:freebsd-doc:
   ```

5. Run `$PREFIX/libexec/gnats/gen-index` to recreate the GNATS index. The output has to be redirected to `$PREFIX/share/gnats/gnats-db/gnats-adm/index`. You can do this periodically from cron(8), or run cvsup(1) from a shell script that does this as well.

   ```
   # /usr/local/libexec/gnats/gen-index \
   > /usr/local/share/gnats/gnats-db/gnats-adm/index
   ```

6. Test the configuration by querying the PR database. This command shows open `docs` PRs.

   ```
   # query-pr -c docs -s open
   ```

7. Pick a PR and close it.

Note: This procedure only works to allow you to view and query the PRs locally. To edit or close them you will still have to log in to `freefall` and do it from there.

9 Who's Who

Besides the repository meisters, there are other FreeBSD project members and teams whom you will probably get to know in your role as a committer. Briefly, and by no means all-inclusively, these are:

Documentation Engineering Team <`doceng@FreeBSD.org`>

> doceng is the group responsible for the documentation build infrastructure, approving new documentation committers, and ensuring that the FreeBSD website and documentation on the FTP site is up to date with respect to the CVS tree. It is not a conflict resolution body. The vast majority of documentation related discussion takes place on the FreeBSD documentation project mailing list (http://lists.FreeBSD.org/mailman/listinfo/freebsd-doc). More details regarding the doceng team can be found in its charter (http://www.FreeBSD.org/internal/doceng.html). Committers interested in contributing to the documentation should familiarize themselves with the Documentation Project Primer (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/fdp-primer/index.html).

Ruslan Ermilov <`ru@FreeBSD.org`>

> Ruslan is Mister mdoc(7). If you are writing a manual page and need some advice on the structure, or the markup, ask Ruslan.

Bruce Evans <`bde@FreeBSD.org`>

Bruce is the Style Police-Meister. When you do a commit that could have been done better, Bruce will be there to tell you. Be thankful that someone is. Bruce is also very knowledgeable on the various standards applicable to FreeBSD.

Konstantin Belousov <`kib@FreeBSD.org`>, Marc Fonvieille <`blackend@FreeBSD.org`>, Josh Paetzel <`jpaetzel@FreeBSD.org`>, Hiroki Sato <`hrs@FreeBSD.org`>, Ken Smith <`kensmith@FreeBSD.org`>, Robert Watson <`rwatson@FreeBSD.org`>, Bjoern A. Zeeb <`bz@FreeBSD.org`>

These are the members of the Release Engineering Team <`re@FreeBSD.org`>. This team is responsible for setting release deadlines and controlling the release process. During code freezes, the release engineers have final authority on all changes to the system for whichever branch is pending release status. If there is something you want merged from FreeBSD-CURRENT to FreeBSD-STABLE (whatever values those may have at any given time), these are the people to talk to about it.

Hiroki is also the keeper of the release documentation (`src/release/doc/*`). If you commit a change that you think is worthy of mention in the release notes, please make sure he knows about it. Better still, send him a patch with your suggested commentary.

Colin Percival <`cperciva@FreeBSD.org`>

Colin is the FreeBSD Security Officer (http://www.FreeBSD.org/security/) and oversees the Security Officer Team <`security-officer@FreeBSD.org`>.

Garrett Wollman <`wollman@FreeBSD.org`>

If you need advice on obscure network internals or are not sure of some potential change to the networking subsystem you have in mind, Garrett is someone to talk to. Garrett is also very knowledgeable on the various standards applicable to FreeBSD.

FreeBSD committer's mailing list

cvs-committers is the entity that the version control system uses to send you all your commit messages. You should *never* send email directly to this list. You should only send replies to this list when they are short and are directly related to a commit.

There is a similar list, svn-committers, which has a similar purpose but is a normal list, i.e. you are free to send any suitable message to this list.

FreeBSD developers mailing list

All committers are subscribed to -developers. This list was created to be a forum for the committers "community" issues. Examples are Core voting, announcements, etc.

The FreeBSD developers mailing list is for the exclusive use of FreeBSD committers. In order to develop FreeBSD, committers must have the ability to openly discuss matters that will be resolved before they are publicly announced. Frank discussions of work in progress are not suitable for open publication and may harm FreeBSD.

All FreeBSD committers are reminded to obey the copyright of the original author(s) of FreeBSD developers mailing list mail. Do not publish or forward messages from the FreeBSD developers mailing list outside the list membership without permission of all of the authors.

Copyright violators will be removed from the FreeBSD developers mailing list, resulting in a suspension of commit privileges. Repeated or flagrant violations may result in permanent revocation of commit privileges.

This list is *not* intended as a place for code reviews or a replacement for the FreeBSD architecture and design mailing list (http://lists.FreeBSD.org/mailman/listinfo/freebsd-arch). In fact using it as such hurts the FreeBSD Project as it gives a sense of a closed list where general decisions affecting all of the FreeBSD using community are made without being "open". Last, but not least *never, never ever, email the FreeBSD developers mailing list and CC:/BCC: another FreeBSD list*. Never, ever email another FreeBSD email list and CC:/BCC: the FreeBSD developers mailing list. Doing so can greatly diminish the benefits of this list.

10 SSH Quick-Start Guide

1. If you do not wish to type your password in every time you use ssh(1), and you use RSA or DSA keys to authenticate, ssh-agent(1) is there for your convenience. If you want to use ssh-agent(1), make sure that you run it before running other applications. X users, for example, usually do this from their `.xsession` or `.xinitrc` file. See ssh-agent(1) for details.
2. Generate a key pair using ssh-keygen(1). The key pair will wind up in your `$HOME/.ssh/` directory.
3. Send your public key (`$HOME/.ssh/id_dsa.pub` or `$HOME/.ssh/id_rsa.pub`) to the person setting you up as a committer so it can be put into the *`yourlogin`* file in `/etc/ssh-keys/` on `freefall`.

Now you should be able to use ssh-add(1) for authentication once per session. This will prompt you for your private key's pass phrase, and then store it in your authentication agent (ssh-agent(1)). If you no longer wish to have your key stored in the agent, issuing `ssh-add -d` will remove it.

Test by doing something such as `ssh freefall.FreeBSD.org ls /usr`.

For more information, see `security/openssh`, ssh(1), ssh-add(1), ssh-agent(1), ssh-keygen(1), and scp(1).

11 Coverity Prevent® Availability for FreeBSD Committers

In January 2006, the FreeBSD Foundation obtained a license for Coverity Prevent® from Coverity® Ltd. With this donation, all FreeBSD developers can obtain access to **Coverity Prevent** analysis results of all FreeBSD Project software.

FreeBSD developers who are interested in obtaining access to the analysis results of the automated **Coverity Prevent** runs, can find out more by logging into `freefall` and reading the relevant bits of the files:

`/usr/local/coverity/coverity_license.txt`

The license terms to which the FreeBSD developers will have to agree in order to use Coverity Prevent analysis results.

`/usr/local/coverity/coverity_announcement.txt`

The announcement posted to the developers' mailing list of the FreeBSD Project. It contains useful information about the FreeBSD Foundation and Coverity Ltd., as well as signup information for registering with the Coverity Prevent installation of the FreeBSD Cluster.

After reading and understanding the license terms of `coverity_license.txt`, all FreeBSD developers who are interested in using the analysis results of Coverity Prevent should read this file.

`/usr/local/coverity/coverity_readme.txt`

A short guide about fixes which are committed to the FreeBSD source tree after being detected by Coverity Prevent and analyzed by a FreeBSD developer.

The FreeBSD Wiki includes a mini-guide for developers who are interested in working with the Coverity Prevent analysis reports: http://wiki.freebsd.org/CoverityPrevent. Please note that this mini-guide is only readable by FreeBSD developers, so if you cannot access this page, you will have to ask someone to add you to the appropriate Wiki access list.

Finally, all FreeBSD developers who are going to use Coverity Prevent are always encouraged to ask for more details and usage information, by posting any questions to the mailing list of the FreeBSD developers.

12 The FreeBSD Committers' Big List of Rules

1. Respect other committers.
2. Respect other contributors.
3. Discuss any significant change *before* committing.
4. Respect existing maintainers (if listed in the `MAINTAINER` field in `Makefile` or in the `MAINTAINER` file in the top-level directory).
5. Any disputed change must be backed out pending resolution of the dispute if requested by a maintainer. Security related changes may override a maintainer's wishes at the Security Officer's discretion.
6. Changes go to FreeBSD-CURRENT before FreeBSD-STABLE unless specifically permitted by the release engineer or unless they are not applicable to FreeBSD-CURRENT. Any non-trivial or non-urgent change which is applicable should also be allowed to sit in FreeBSD-CURRENT for at least 3 days before merging so that it can be given sufficient testing. The release engineer has the same authority over the FreeBSD-STABLE branch as outlined for the maintainer in rule #5.
7. Do not fight in public with other committers; it looks bad. If you must "strongly disagree" about something, do so only in private.
8. Respect all code freezes and read the `committers` and `developers` mailing lists in a timely manner so you know when a code freeze is in effect.
9. When in doubt on any procedure, ask first!
10. Test your changes before committing them.
11. Do not commit to anything under the `src/contrib`, `src/crypto`, or `src/sys/contrib` trees without *explicit* approval from the respective maintainer(s).

As noted, breaking some of these rules can be grounds for suspension or, upon repeated offense, permanent removal of commit privileges. Individual members of core have the power to temporarily suspend commit privileges until core as a whole has the chance to review the issue. In case of an "emergency" (a committer doing damage to the repository), a temporary suspension may also be done by the repository meisters. Only a 2/3 majority of core has the authority to suspend commit privileges for longer than a week or to remove them permanently. This rule does not exist to set core up as a bunch of cruel dictators who can dispose of committers as casually as empty soda cans, but to give the project a kind of safety fuse. If someone is out of control, it is important to be able to deal with this immediately rather than be paralyzed by debate. In all cases, a committer whose privileges are suspended or revoked is entitled to a "hearing" by core, the total duration of the suspension being determined at that time. A committer whose privileges are suspended may also request a review of the decision after 30 days and every 30 days thereafter (unless the total suspension period is less than 30 days). A committer whose privileges have been revoked entirely may request a review after a period of 6 months has elapsed. This review policy is *strictly informal* and, in all cases, core reserves the right to either act on or disregard requests for review if they feel their original decision to be the right one.

In all other aspects of project operation, core is a subset of committers and is bound by the *same rules*. Just because someone is in core this does not mean that they have special dispensation to step outside any of the lines painted here; core's "special powers" only kick in when it acts as a group, not on an individual basis. As individuals, the core team members are all committers first and core second.

12.1 Details

1. Respect other committers.

   This means that you need to treat other committers as the peer-group developers that they are. Despite our occasional attempts to prove the contrary, one does not get to be a committer by being stupid and nothing rankles more than being treated that way by one of your peers. Whether we always feel respect for one another or not (and everyone has off days), we still have to *treat* other committers with respect at all times, on public forums and in private email.

   Being able to work together long term is this project's greatest asset, one far more important than any set of changes to the code, and turning arguments about code into issues that affect our long-term ability to work harmoniously together is just not worth the trade-off by any conceivable stretch of the imagination.

   To comply with this rule, do not send email when you are angry or otherwise behave in a manner which is likely to strike others as needlessly confrontational. First calm down, then think about how to communicate in the most effective fashion for convincing the other person(s) that your side of the argument is correct, do not just blow off some steam so you can feel better in the short term at the cost of a long-term flame war. Not only is this very bad "energy economics", but repeated displays of public aggression which impair our ability to work well together will be dealt with severely by the project leadership and may result in suspension or termination of your commit privileges. The project leadership will take into account both public and private communications brought before it. It will not seek the disclosure of private communications, but it will take it into account if it is volunteered by the committers involved in the complaint.

   All of this is never an option which the project's leadership enjoys in the slightest, but unity comes first. No amount of code or good advice is worth trading that away.

2. Respect other contributors.

   You were not always a committer. At one time you were a contributor. Remember that at all times. Remember what it was like trying to get help and attention. Do not forget that your work as a contributor was very important to you. Remember what it was like. Do not discourage, belittle, or demean contributors. Treat them with respect. They are our committers in waiting. They are every bit as important to the project as committers. Their contributions are as valid and as important as your own. After all, you made many contributions before you became a committer. Always remember that.

   Consider the points raised under 1 and apply them also to contributors.

3. Discuss any significant change *before* committing.

The repository is not where changes should be initially submitted for correctness or argued over, that should happen first in the mailing lists and the commit should only happen once something resembling consensus has been reached. This does not mean that you have to ask permission before correcting every obvious syntax error or manual page misspelling, simply that you should try to develop a feel for when a proposed change is not quite such a no-brainer and requires some feedback first. People really do not mind sweeping changes if the result is something clearly better than what they had before, they just do not like being *surprised* by those changes. The very best way of making sure that you are on the right track is to have your code reviewed by one or more other committers.

When in doubt, ask for review!

4. Respect existing maintainers if listed.

   Many parts of FreeBSD are not "owned" in the sense that any specific individual will jump up and yell if you commit a change to "their" area, but it still pays to check first. One convention we use is to put a maintainer line in the `Makefile` for any package or subtree which is being actively maintained by one or more people; see http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/developers-handbook/policies.html (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/developers-handbook/policies.html) for documentation on this. Where sections of code have several maintainers, commits to affected areas by one maintainer need to be reviewed by at least one other maintainer. In cases where the "maintainer-ship" of something is not clear, you can also look at the repository logs for the file(s) in question and see if someone has been working recently or predominantly in that area.

   Other areas of FreeBSD fall under the control of someone who manages an overall category of FreeBSD evolution, such as internationalization or networking. See http://www.FreeBSD.org/administration.html (http://www.FreeBSD.org/administration.html) for more information on this.

5. Any disputed change must be backed out pending resolution of the dispute if requested by a maintainer. Security related changes may override a maintainer's wishes at the Security Officer's discretion.

   This may be hard to swallow in times of conflict (when each side is convinced that they are in the right, of course) but a version control system makes it unnecessary to have an ongoing dispute raging when it is far easier to simply reverse the disputed change, get everyone calmed down again and then try to figure out what is the best way to proceed. If the change turns out to be the best thing after all, it can be easily brought back. If it turns out not to be, then the users did not have to live with the bogus change in the tree while everyone was busily debating its merits. People *very* rarely call for back-outs in the repository since discussion generally exposes bad or controversial changes before the commit even happens, but on such rare occasions the back-out should be done without argument so that we can get immediately on to the topic of figuring out whether it was bogus or not.

6. Changes go to FreeBSD-CURRENT before FreeBSD-STABLE unless specifically permitted by the release engineer or unless they are not applicable to FreeBSD-CURRENT. Any non-trivial or non-urgent change which is applicable should also be allowed to sit in FreeBSD-CURRENT for at least 3 days before merging so that it can be given sufficient testing. The release engineer has the same authority over the FreeBSD-STABLE branch as outlined in rule #5.

   This is another "do not argue about it" issue since it is the release engineer who is ultimately responsible (and gets beaten up) if a change turns out to be bad. Please respect this and give the release engineer your full cooperation when it comes to the FreeBSD-STABLE branch. The management of FreeBSD-STABLE may frequently seem to be overly conservative to the casual observer, but also bear in mind the fact that conservatism is supposed to be the hallmark of FreeBSD-STABLE and different rules apply there than in FreeBSD-CURRENT. There is also really no point in having FreeBSD-CURRENT be a testing ground if changes are merged over to FreeBSD-STABLE immediately. Changes need a chance to be tested by the FreeBSD-CURRENT developers, so allow some time to elapse before merging unless the FreeBSD-STABLE fix is critical, time sensitive or so obvious as to make further testing unnecessary (spelling fixes to manual pages, obvious bug/typo fixes, etc.) In other words, apply common sense.

Changes to the security branches (for example, `RELENG_7_0`) must be approved by a member of the Security Officer Team <`security-officer@FreeBSD.org`>, or in some cases, by a member of the Release Engineering Team <`re@FreeBSD.org`>.

7. Do not fight in public with other committers; it looks bad. If you must "strongly disagree" about something, do so only in private.

   This project has a public image to uphold and that image is very important to all of us, especially if we are to continue to attract new members. There will be occasions when, despite everyone's very best attempts at self-control, tempers are lost and angry words are exchanged. The best thing that can be done in such cases is to minimize the effects of this until everyone has cooled back down. That means that you should not air your angry words in public and you should not forward private correspondence to public mailing lists or aliases. What people say one-to-one is often much less sugar-coated than what they would say in public, and such communications therefore have no place there - they only serve to inflame an already bad situation. If the person sending you a flame-o-gram at least had the grace to send it privately, then have the grace to keep it private yourself. If you feel you are being unfairly treated by another developer, and it is causing you anguish, bring the matter up with core rather than taking it public. Core will do its best to play peace makers and get things back to sanity. In cases where the dispute involves a change to the codebase and the participants do not appear to be reaching an amicable agreement, core may appoint a mutually-agreeable 3rd party to resolve the dispute. All parties involved must then agree to be bound by the decision reached by this 3rd party.

8. Respect all code freezes and read the `committers` and `developers` mailing list on a timely basis so you know when a code freeze is in effect.

   Committing unapproved changes during a code freeze is a really big mistake and committers are expected to keep up-to-date on what is going on before jumping in after a long absence and committing 10 megabytes worth of accumulated stuff. People who abuse this on a regular basis will have their commit privileges suspended until they get back from the FreeBSD Happy Reeducation Camp we run in Greenland.

9. When in doubt on any procedure, ask first!

   Many mistakes are made because someone is in a hurry and just assumes they know the right way of doing something. If you have not done it before, chances are good that you do not actually know the way we do things and really need to ask first or you are going to completely embarrass yourself in public. There is no shame in asking "how in the heck do I do this?" We already know you are an intelligent person; otherwise, you would not be a committer.

10. Test your changes before committing them.

    This may sound obvious, but if it really were so obvious then we probably would not see so many cases of people clearly not doing this. If your changes are to the kernel, make sure you can still compile both GENERIC and LINT. If your changes are anywhere else, make sure you can still make world. If your changes are to a branch, make sure your testing occurs with a machine which is running that code. If you have a change which also may break another architecture, be sure and test on all supported architectures. Please refer to the FreeBSD Internal Page (http://www.FreeBSD.org/internal/) for a list of available resources. As other architectures are added to the FreeBSD supported platforms list, the appropriate shared testing resources will be made available.

11. Do not commit to anything under the `src/contrib`, `src/crypto`, and `src/sys/contrib` trees without *explicit* approval from the respective maintainer(s).

    The trees mentioned above are for contributed software usually imported onto a vendor branch. Committing something there, even if it does not take the file off the vendor branch, may cause unnecessary headaches for those responsible for maintaining that particular piece of software. Thus, unless you have *explicit* approval from the maintainer (or you are the maintainer), do *not* commit there!

Please note that this does not mean you should not try to improve the software in question; you are still more than welcome to do so. Ideally, you should submit your patches to the vendor. If your changes are FreeBSD-specific, talk to the maintainer; they may be willing to apply them locally. But whatever you do, do *not* commit there by yourself!

Contact the FreeBSD core team if you wish to take up maintainership of an unmaintained part of the tree.

12.2 Policy on Multiple Architectures

FreeBSD has added several new architecture ports during recent release cycles and is truly no longer an i386™ centric operating system. In an effort to make it easier to keep FreeBSD portable across the platforms we support, core has developed the following mandate:

> Our 32-bit reference platform is i386, and our 64-bit reference platform is sparc64. Major design work (including major API and ABI changes) must prove itself on at least one 32-bit and at least one 64-bit platform, preferably the primary reference platforms, before it may be committed to the source tree.

The i386 and sparc64 platforms were chosen due to being more readily available to developers and as representatives of more diverse processor and system designs - big vs little endian, register file vs register stack, different DMA and cache implementations, hardware page tables vs software TLB management etc.

The ia64 platform has many of the same complications that sparc64 has, but is still limited in availability to developers.

We will continue to re-evaluate this policy as cost and availability of the 64-bit platforms change.

Developers should also be aware of our Tier Policy for the long term support of hardware architectures. The rules here are intended to provide guidance during the development process, and are distinct from the requirements for features and architectures listed in that section. The Tier rules for feature support on architectures at release-time are more strict than the rules for changes during the development process.

12.3 Other Suggestions

When committing documentation changes, use a spell checker before committing. For all SGML docs, you should also verify that your formatting directives are correct by running `make lint`.

For all on-line manual pages, run `manck` (from ports) over the manual page to verify all of the cross references and file references are correct and that the man page has all of the appropriate `MLINKs` installed.

Do not mix style fixes with new functionality. A style fix is any change which does not modify the functionality of the code. Mixing the changes obfuscates the functionality change when asking for differences between revisions, which can hide any new bugs. Do not include whitespace changes with content changes in commits to `doc/` or `www/`. The extra clutter in the diffs makes the translators' job much more difficult. Instead, make any style or whitespace changes in separate commits that are clearly labeled as such in the commit message.

12.4 Deprecating Features

When it is necessary to remove functionality from software in the base system the following guidelines should be followed whenever possible:

1. Mention is made in the manual page and possibly the release notes that the option, utility, or interface is deprecated. Use of the deprecated feature generates a warning.
2. The option, utility, or interface is preserved until the next major (point zero) release.
3. The option, utility, or interface is removed and no longer documented. It is now obsolete. It is also generally a good idea to note its removal in the release notes.

13 Support for Multiple Architectures

FreeBSD is a highly portable operating system intended to function on many different types of hardware architectures. Maintaining clean separation of Machine Dependent (MD) and Machine Independent (MI) code, as well as minimizing MD code, is an important part of our strategy to remain agile with regards to current hardware trends. Each new hardware architecture supported by FreeBSD adds substantially to the cost of code maintenance, toolchain support, and release engineering. It also dramatically increases the cost of effective testing of kernel changes. As such, there is strong motivation to differentiate between classes of support for various architectures while remaining strong in a few key architectures that are seen as the FreeBSD "target audience".

13.1 Statement of General Intent

The FreeBSD Project targets "production quality commercial off-the-shelf (COTS) workstation, server, and high-end embedded systems". By retaining a focus on a narrow set of architectures of interest in these environments, the FreeBSD Project is able to maintain high levels of quality, stability, and performance, as well as minimize the load on various support teams on the project, such as the ports team, documentation team, security officer, and release engineering teams. Diversity in hardware support broadens the options for FreeBSD consumers by offering new features and usage opportunities (such as support for 64-bit CPUs, use in embedded environments, etc.), but these benefits must always be carefully considered in terms of the real-world maintenance cost associated with additional platform support.

The FreeBSD Project differentiates platform targets into four tiers. Each tier includes a specification of the requirements for an architecture to be in that tier, as well as specifying the obligations of developers with regards to the platform. In addition, a policy is defined regarding the circumstances required to change the tier of an architecture.

13.2 Tier 1: Fully Supported Architectures

Tier 1 platforms are fully supported by the security officer, release engineering, and toolchain maintenance staff. New features added to the operating system must be fully functional across all Tier 1 architectures for every release (features which are inherently architecture-specific, such as support for hardware device drivers, may be exempt from this requirement). In general, all Tier 1 platforms must have build and Tinderbox support either in the FreeBSD.org cluster, or be easily available for all developers. Embedded platforms may substitute an emulator available in the FreeBSD cluster for actual hardware.

Tier 1 architectures are expected to be Production Quality with respects to all aspects of the FreeBSD operating system, including installation and development environments.

Tier 1 architectures are expected to be completely integrated into the source tree and have all features necessary to produce an entire system relevant for that target architecture. Tier 1 architectures generally have at least 6 active developers.

Tier 1 architectures are expected to be fully supported by the ports system. All the ports should build on a Tier 1 platform, or have the appropriate filters to prevent the inappropriate ones from building there. The packaging system must support all Tier 1 architectures. To ensure an architecture's Tier 1 status, proponents of that architecture must show that all relevant packages can be built on that platform.

Tier 1 embedded architectures must be able to cross-build packages on at least one other Tier 1 architecture. The packages must be the most relevant for the platform, but may be a non-empty subset of those that build natively.

Tier 1 architectures must be fully documented. All basic operations need to be covered by the handbook or other documents. All relevant integration documentation must also be integrated into the tree, or readily available.

Current Tier 1 platforms are i386 and amd64.

13.3 Tier 2: Developmental Architectures

Tier 2 platforms are not supported by the security officer and release engineering teams. Platform maintainers are responsible for toolchain support in the tree. The toolchain maintainer is expected to work with the platform maintainers to refine these changes. Major new toolchain components are allowed to break support for Tier 2 architectures if the FreeBSD-local changes have not been incorporated upstream. The toolchain maintainers are expected to provide prompt review of any proposed changes and cannot block, through their inaction, changes going into the tree. New features added to FreeBSD should be feasible to implement on these platforms, but an implementation is not required before the feature may be added to the FreeBSD source tree. New features that may be difficult to implement on Tier 2 architectures should provide a means of disabling them on those architectures. The implementation of a Tier 2 architecture may be committed to the main FreeBSD tree as long as it does not interfere with production work on Tier 1 platforms, or substantially with other Tier 2 platforms. Before a Tier 2 platform can be added to the FreeBSD base source tree, the platform must be able to boot multi-user on actual hardware. Generally, there must be at least three active developers working on the platform.

Tier 2 architectures are usually systems targeted at Tier 1 support, but that are still under development. Architectures reaching end of life may also be moved from Tier 1 status to Tier 2 status as the availability of resources to continue to maintain the system in a Production Quality state diminishes. Well supported niche architectures may also be Tier 2.

Tier 2 architectures may have some support for them integrated into the ports infrastructure. They may have cross compilation support added, at the discretion of portmgr. Some ports must built natively into packages if the package system supports that architecture. If not integrated into the base system, some external patches for the architecture for ports must be available.

Tier 2 architectures can be integrated into the FreeBSD handbook. The basics for how to get a system running must be documented, although not necessarily for every single board or system a Tier 2 architecture supports. The supported hardware list must exist and should be no more than a couple of months old. It should be integrated into the FreeBSD documentation.

Current Tier 2 platforms are arm, ia64, pc98, powerpc, and sparc64.

13.4 Tier 3: Experimental Architectures

Tier 3 platforms are not supported by the security officer and release engineering teams. At the discretion of the toolchain maintainer, they may be supported in the toolchain. Tier 3 platforms are architectures in the early stages of development, for non-mainstream hardware platforms, or which are considered legacy systems unlikely to see broad future use. New Tier 3 systems will not be committed to the base source tree. Support for Tier 3 systems may be worked on in the FreeBSD Perforce Repository, providing source control and easier change integration from the main FreeBSD tree. Platforms that transition to Tier 3 status may be removed from the tree if they are no longer actively supported by the FreeBSD developer community at the discretion of the release engineer.

Tier 3 platforms may have ports support, either integrated or external, but do not require it.

Tier 3 platforms must have the basics documented for how to build a kernel and how to boot it on at least one target hardware or emulation environment. This documentation need not be integrated into the FreeBSD tree.

Current Tier 3 platforms are mips and S/390®.

13.5 Tier 4: Unsupported Architectures

Tier 4 systems are not supported in any form by the project.

All systems not otherwise classified into a support tier are Tier 4 systems.

13.6 Policy on Changing the Tier of an Architecture

Systems may only be moved from one tier to another by approval of the FreeBSD Core Team, which shall make that decision in collaboration with the Security Officer, Release Engineering, and toolchain maintenance teams.

14 Ports Specific FAQ

1. Adding a New Port

1.1. How do I add a new port?

First, please read the section about repository copies.

The easiest way to add a new port is to use the `addport` script from your machine (located in the `ports/Tools/scripts` directory). It will add a port from the directory you specify, determining the category automatically from the port `Makefile`. It will also add an entry to the port's category `Makefile`. It was written by Michael Haro <`mharo@FreeBSD.org`> and Will Andrews <`will@FreeBSD.org`>, and is currently maintained by Renato Botelho <`garga@FreeBSD.org`>, so please send questions/patches about `addport` to him.

1.2. Any other things I need to know when I add a new port?

Check the port, preferably to make sure it compiles and packages correctly. This is the recommended sequence:

```
# make install
# make package
# make deinstall
# pkg_add package you built above
# make deinstall
# make reinstall
# make package
```

The Porters Handbook (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/porters-handbook/index.html) contains more detailed instructions.

Use portlint(1) to check the syntax of the port. You do not necessarily have to eliminate all warnings but make sure you have fixed the simple ones.

If the port came from a submitter who has not contributed to the Project before, add that person's name to the Additional Contributors (http://www.FreeBSD.org/doc/en_US.ISO8859-1/articles/contributors/contrib-additional.html) section of the FreeBSD Contributors List.

Close the PR if the port came in as a PR. To close a PR, just do **`edit-pr`** ***`PR#`*** on `freefall` and change the `state` from `open` to `closed`. You will be asked to enter a log message and then you are done.

2. Removing an Existing Port

2.1. How do I remove an existing port?

First, please read the section about repository copies. Before you remove the port, you have to verify there are no other ports depending on it.

- Make sure there is no dependency on the port in the ports collection:
  - The port's PKGNAME should appear in exactly one line in a recent INDEX file.
  - No other ports should contain any reference to the port's directory or PKGNAME in their Makefiles
- Then, remove the port:
  1. Remove the port's files via `cvs remove`.
  2. Remove the `SUBDIR` listing of the port in the parent directory `Makefile`.
  3. Add an entry to `ports/MOVED`.
  4. Remove the port from `ports/LEGAL` if it is there.

Alternatively, you can use the `rmport` script, from `ports/Tools/scripts`. This script has been written by Vasil Dimov <`vd@FreeBSD.org`>, who is also its current maintainer, so please send questions, patches or suggestions about `rmport` to him.

3. Re-adding a Deleted Port

3.1. How do I re-add a deleted port?

This is essentially the reverse of deleting a port.

1. Figure out when the port was removed. Use the ports cvsweb (http://www.freebsd.org/cgi/cvsweb.cgi/ports/) and then navigate to *`category`*/*`portname`*/`Attic`/ . Pick a date that is before the removal but after the last true commit.
2. In the proper directory: `cvs update -D` *`datespec`*.
3. Perform whatever changes are necessary to make the port work again. If it was deleted because the distfiles are no longer available you will need to volunteer to host them yourself, or find someone else to do so.
4. `cvs add` the updated files.
5. Restore the `SUBDIR` listing of the port in the parent directory `Makefile`, and delete the entry from `ports/MOVED`.
6. If the port had an entry in `ports/LEGAL`, restore it.
7. `cvs commit` these changes, preferably in one step.

4. Repository Copies

4.1. When do we need a repository copy?

When you want to add a port that is related to any port that is already in the tree in a separate directory, you have to do a repository copy. Here *related* means it is a different version or a slightly modified version. Examples are `print/ghostscript*` (different versions) and `x11-wm/windowmaker*` (English-only and internationalized version).

Another example is when a port is moved from one subdirectory to another, or when you want to change the name of a directory because the author(s) renamed their software even though it is a descendant of a port already in a tree.

4.2. When do we *not* need a repository copy?

When there is no history to preserve. If a port is added into a wrong category and is moved immediately, it suffices to simply `cvs remove` the old one and `addport` the new one.

4.3. What do I need to do?

File a PR in **GNATS**, listing the reasons for the repository copy request. Assign it to `portmgr` and set `state` to `repocopy`. In a few days, `portmgr` will do a repository copy from the old to the new location, and reassign the PR back to you. Once everything is done, perform the following:

- When a port has been repo copied:

  1. Do a force commit on the files of the copied port, stating repository copy was performed.
  2. Upgrade the copied port to the new version. Remember to change the `LATEST_LINK` so there are no duplicate ports with the same name. In some rare cases it may be necessary to change the `PORTNAME`

   instead of `LATEST_LINK`, but this should only be done when it is really needed — e.g. using an existing port as the base for a very similar program with a different name, or upgrading a port to a new upstream version which actually changes the distribution name, like the transition from `textproc/libxml` to `textproc/libxml2`. In most cases, changing `LATEST_LINK` should suffice.

3. Add the new subdirectory to the `SUBDIR` listing in the parent directory `Makefile`. You can run `make checksubdirs` in the parent directory to check this.
4. If the port changed categories, modify the `CATEGORIES` line of the port's `Makefile` accordingly
5. Add an entry to `ports/MOVED`, if you remove the original port.

- When removing a port:

    1. Perform a thorough check of the ports collection for any dependencies on the old port location/name, and update them. Running `grep` on `INDEX` is not enough because some ports have dependencies enabled by compile-time options. A full `grep -r` of the ports collection is recommended.
    2. Remove the old port and the old `SUBDIR` entry.
    3. Add an entry to `ports/MOVED`.

- After repo moves ("rename" operations where a port is copied and the old location is removed):

    1. Follow the same steps that are outlined in the previous two entries, to activate the new location of the port and remove the old one.

5. Ports Freeze

5.1. What is a "ports freeze"?

Before a release, it is necessary to restrict commits to the ports tree for a short period of time while the packages and the release itself are being built. This is to ensure consistency among the various parts of the release, and is called the "ports freeze".

For more information on the background and policies surrounding a ports freeze, see the Portmgr Quality Assurance page (http://www.FreeBSD.org/portmgr/qa.html).

5.2. What is a "ports slush" or "feature freeze"?

During a release cycle the ports tree may be in a "slush" state instead of in a hard freeze. The goal during a slush is to reach a stable ports tree to avoid rebuilding large sets of packages for the release and to tag the tree. During this time "sweeping changes" are prohibited unless specifically permitted by portmgr. Complete details about what qualifies as a sweeping change can be found on the Portmgr Implementation page (http://www.FreeBSD.org/portmgr/implementation.html).

The benefit of a slush as opposed to a complete freeze is that it allows maintainers to continue adding new ports, making routine version updates, and bug fixes to most existing ports, as long as the number of affected ports is minimal. For example, updating the shared library version on a port that many other ports depend on.

5.3. How long is a ports freeze or slush?

A freeze only lasts long enough to tag the tree. A slush usually lasts a week or two, but may last longer.

5.4. What does it mean to me?

During a ports freeze, you are not allowed to commit anything to the tree without explicit approval from the Ports Management Team. "Explicit approval" here means that you send a patch to the Ports Management Team for review and get a reply saying, "Go ahead and commit it."

Not everything is allowed to be committed during a freeze. Please see the Portmgr Quality Assurance page (http://www.FreeBSD.org/portmgr/qa.html) for more information.

Note that you do not have implicit permission to fix a port during the freeze just because it is broken.

During a ports slush, you are still allowed to commit but you must exercise more caution in what you commit. Furthermore a special note (typically "Feature Safe: yes") must be added to the commit message.

5.5. How do I know when the ports slush starts?

The Ports Management Team will send out warning messages to the FreeBSD ports mailing list (http://lists.FreeBSD.org/mailman/listinfo/freebsd-ports) and FreeBSD committer's mailing list announcing the start of the impending release, usually two or three weeks in advance. The exact starting time will not be determined until a few days before the actual release. This is because the ports slush has to be synchronized with the release, and it is usually not known until then when exactly the release will be rolled.

When the slush starts, there will be another announcement to the FreeBSD ports mailing list (http://lists.FreeBSD.org/mailman/listinfo/freebsd-ports) and FreeBSD committer's mailing list, of course.

5.6. How do I know when the freeze or slush ends?

A few hours after the release, the Ports Management Team will send out a mail to the FreeBSD ports mailing list (http://lists.FreeBSD.org/mailman/listinfo/freebsd-ports) and FreeBSD committer's mailing list announcing the end of the ports freeze or slush. Note that the release being cut does not automatically indicate the end of the freeze. We have to make sure there will be no last minute snafus that result in an immediate re-rolling of the release.

6. Creating a New Category

6.1. What is the procedure for creating a new category?

Please see Proposing a New Category (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/porters-handbook/makefile-categories.html#PROPOSING-CATEGORIES) in the Porter's Handbook. Once that procedure has been followed and the PR has been assigned to Ports Management Team <`portmgr@FreeBSD.org`>, it is their decision whether or not to approve it. If they do, it is their responsibility to do the following:

1. Perform any needed repocopies. (This only applies to physical categories.)
2. Update the `VALID_CATEGORIES` definition in `ports/Mk/bsd.port.mk`.
3. Assign the PR back to you.

6.2. What do I need to do to implement a new physical category?

The procedure is a strict superset of the one to repocopy individual ports (see above).

1. Upgrade each copied port's `Makefile`. Do not connect the new category to the build yet.

   To do this, you will need to:

   1. Change the port's `CATEGORIES` (this was the point of the exercise, remember?) The new category should be listed *first*. This will help to ensure that the `PKGORIGIN` is correct.
   2. Run a `make describe`. Since the top-level `make index` that you will be running in a few steps is an iteration of `make describe` over the entire ports hierarchy, catching any errors here will save you having to re-run that step later on.
   3. If you want to be really thorough, now might be a good time to run portlint(1).

2. Check that the `PKGORIGIN`s are correct. The ports system uses each port's `CATEGORIES` entry to create its `PKGORIGIN`, which is used to connect installed packages to the port directory they were built from. If this entry is wrong, common port tools like pkg_version(1) and portupgrade(1) fail.

   To do this, use the `chkorigin.sh` tool, as follows: `env PORTSDIR=`*`/path/to/ports`*` sh -e `*`/path/to/ports`*`/Tools/scripts/chkorigin.sh` . This will check *every* port in the ports tree, even those not connected to the build, so you can run it directly after the repocopy. Hint: do not forget to look at the `PKGORIGIN`s of any slave ports of the ports you just repocopied!
3. On your own local system, test the proposed changes: first, comment out the `SUBDIR` entries in the old ports' categories' `Makefile`s; then enable building the new category in `ports/Makefile`. Run `make checksubdirs` in the affected category directories to check the `SUBDIR` entries. Next, in the `ports/` directory, run `make index`. This can take over 40 minutes on even modern systems; however, it is a necessary step to prevent problems for other people.
4. Once this is done, you can commit the updated `ports/Makefile` to connect the new category to the build and also commit the `Makefile` changes for the old category or categories.
5. Add appropriate entries to `ports/MOVED`.
6. Update the instructions for cvsup(1):

   - add the category to `distrib/cvsup/sup/README`
   - adding the following files into `distrib/cvsup/sup/ports-`*`categoryname`*: `list.cvs` and `releases`.
   - add the category to `src/share/examples/cvsup/ports-supfile`

   (Note: these are in the src, not the ports, repository). If you are not a src committer, you will need to submit a PR for this.
7. Update the list of categories used by sysinstall(8) in `src/usr.sbin/sysinstall`.

8. Update the documentation by modifying the following:
   - the section of the Handbook that lists the cvsup collections (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/handbook/cvsup.html#CVSUP-COLLEC).
   - the list of categories (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/porters-handbook/makefile-categories.html#PORTING-CATEGORIES) in the Porter's Handbook
   - `www/en/ports/categories`. Note that these are now displayed by sub-groups, as specified in `www/en/ports/categories.descriptions`.

   (Note: these are in the docs, not the ports, repository). If you are not a docs committer, you will need to submit a PR for this.
9. Only once all the above have been done, and no one is any longer reporting problems with the new ports, should the old ports be deleted from their previous locations in the repository.

It is not necessary to manually update the ports web pages (http://www.FreeBSD.org/ports/index.html) to reflect the new category. This is now done automatically via your change to `www/en/ports/categories` and the daily automated rebuild of `INDEX`.

6.3. What do I need to do to implement a new virtual category?

This is much simpler than a physical category. You only need to modify the following:

- `src/usr.sbin/sysinstall`
- the list of categories (http://www.FreeBSD.org/doc/en_US.ISO8859-1/books/porters-handbook/makefile-categories.html#PORTING-CATEGORIES) in the Porter's Handbook
- `www/en/ports/categories`

7. Miscellaneous Questions

7.1. How do I know if my port is building correctly or not?

First, go check http://pointyhat.FreeBSD.org/errorlogs/. There you will find error logs from the latest package building runs on all supported platforms for the most recent branches.

However, just because the port does not show up there does not mean it is building correctly. (One of the dependencies may have failed, for instance.) The relevant directories are available on `pointyhat` under `/a/portbuild/<arch>/<major_version>` so feel free to dig around. Each architecture and version has the following subdirectories:

```
errors        error logs from latest <major_version> run on <arch>
logs          all logs from latest <major_version> run on <arch>
packages      packages from latest <major_version> run on <arch>
bak/errors    error logs from last complete <major_version> run on <arch>
bak/logs      all logs from last complete <major_version> run on <arch>
bak/packages  packages from last complete <major_version> run on <arch>
```

Basically, if the port shows up in `packages`, or it is in `logs` but not in `errors`, it built fine. (The `errors` directories are what you get from the web page.)

7.2. I added a new port. Do I need to add it to the `INDEX`?

No, `INDEX` is no longer stored in the CVS repository. The file can either be generated by running `make index`, or a pre-generated version can be downloaded with `make fetchindex`.

7.3. Are there any other files I am not allowed to touch?

Any file directly under `ports/`, or any file under a subdirectory that starts with an uppercase letter (`Mk/`, `Tools/`, etc.). In particular, the Ports Management Team is very protective of `ports/Mk/bsd.port*.mk` so do not commit changes to those files unless you want to face his wra(i)th.

7.4. What is the proper procedure for updating the checksum for a port's distfile when the file changes without a version change?

When the checksum for a port's distfile is updated due to the author updating the file without changing the port's revision, the commit message should include a summary of the relevant diffs between the original and new distfile to ensure that the distfile has not been corrupted or maliciously altered. If the current version of the port has been in the ports tree for a while, a copy of the old distfile will usually be available on the ftp servers; otherwise the author or maintainer should be contacted to find out why the distfile has changed.

15 Issues Specific To Developers Who Are Not Committers

A few people who have access to the FreeBSD machines do not have commit bits. For instance, the project is willing to give access to the GNATS database to contributors who have shown interest and dedication in working on Problem Reports.

Almost all of this document will apply to these developers as well (except things specific to commits and the mailing list memberships that go with them). In particular, we recommend that you read:

- Administrative Details
- Conventions

  Note: You should get your mentor to add you to the "Additional Contributors" (`doc/en_US.ISO8859-1/articles/contributors/contrib.additional.sgml`), if you are not already listed there.

- Developer Relations
- SSH Quick-Start Guide
- The FreeBSD Committers' Big List of Rules

16 Perks of the Job

Unfortunately, there are not many perks involved with being a committer. Recognition as a competent software engineer is probably the only thing that will be of benefit in the long run. However, there are at least some perks:

Direct access to `cvsup-master`

As a committer, you may apply to Jun Kuriyama <`kuriyama@FreeBSD.org`> for direct access to `cvsup-master.FreeBSD.org`, providing the public key output from `cvpasswd` *`yourusername`*`@FreeBSD.org freefall.FreeBSD.org`. Please note: you must specify `freefall.FreeBSD.org` on the `cvpasswd` command line even though the actual server is `cvsup-master`. Access to `cvsup-master` should not be overused as it is a busy machine.

Free 4-CD and DVD Sets

FreeBSD committers can get a free 4-CD or DVD set at conferences from FreeBSD Mall, Inc. (http://www.freebsdmall.com). The sets are no longer available as a subscription due to the high shipment costs to countries outside the USA.

Freenode IRC Cloaks

FreeBSD developers may request a cloaked hostmask for their account on the Freenode IRC network in the form of `freebsd/developer/`*`freefall name`* or `freebsd/developer/`*`NickServ name`*. To request a cloak, send an email to Eitan Adler <`eadler@FreeBSD.org`> with your requested hostmask and NickServ account name.

17 Miscellaneous Questions

1. Why are trivial or cosmetic changes to files on a vendor branch a bad idea?

- From now on, every new vendor release of that file will need to have patches merged in by hand.
- From now on, every new vendor release of that file will need to have patches *verified* by hand.
- The `-j` option does not work very well. Ask David O'Brien <`obrien@FreeBSD.org`> for horror stories.

2. How do I add a new file to a branch?

To add a file onto a branch, simply checkout or update to the branch you want to add to and then add the file using the add operation as you normally would. This works fine for the `doc` and `ports` trees. The `src` tree uses SVN and requires more care because of the `mergeinfo` properties. See section 1.4.6 of the Subversion Primer (http://wiki.freebsd.org/SubversionPrimer) for details. Refer to SubversionPrimer/Merging (http://wiki.freebsd.org/SubversionPrimer/Merging) for details on how to perform an MFC.

3. What "meta" information should I include in a commit message?

As well as including an informative message with each commit you may need to include some additional information as well.

This information consists of one or more lines containing the key word or phrase, a colon, tabs for formatting, and then the additional information.

The key words or phrases are:

`PR:`	The problem report (if any) which is affected (typically, by being closed) by this commit.
`Submitted by:`	The name and e-mail address of the person that submitted the fix; for committers, just the username on the FreeBSD cluster.
`Reviewed by:`	The name and e-mail address of the person or people that reviewed the change; for committers, just the username on the FreeBSD cluster. If a patch was submitted to a mailing list for review, and the review was favorable, then just include the list name.
`Approved by:`	The name and e-mail address of the person or people that approved the change; for committers, just the username on the FreeBSD cluster. It is customary to get prior approval for a commit if it is to an area of the tree to which you do not usually commit. In addition, during the run up to a new release all commits *must* be approved by the release engineering team. If these are your first commits then you should have passed them past your mentor first, and you should list your mentor, as in "*`username-of-mentor`* `(mentor)`".
`Obtained from:`	The name of the project (if any) from which the code was obtained.
`MFC after:`	If you wish to receive an e-mail reminder to MFC at a later date, specify the number of days, weeks, or months after which an MFC is planned.
`Security:`	If the change is related to a security vulnerability or security exposure, include one or more references or a description of the issue.

Example 1. Commit log for a commit based on a PR

You want to commit a change based on a PR submitted by John Smith containing a patch. The end of the commit message should look something like this.

```
...

PR:               foo/12345
Submitted by:     John Smith <John.Smith@example.com>
```

Example 2. Commit log for a commit needing review

You want to change the virtual memory system. You have posted patches to the appropriate mailing list (in this case, `freebsd-arch`) and the changes have been approved.

```
...

Reviewed by:      -arch
```

Example 3. Commit log for a commit needing approval

You want to commit a change to a section of the tree with a MAINTAINER assigned. You have collaborated with the listed MAINTAINER, who has told you to go ahead and commit.

```
...

Approved by:        abc
```

Where *`abc`* is the account name of the person who approved.

Example 4. Commit log for a commit bringing in code from OpenBSD

You want to commit some code based on work done in the OpenBSD project.

```
...

Obtained from:      OpenBSD
```

Example 5. Commit log for a change to FreeBSD-CURRENT with a planned commit to FreeBSD-STABLE to follow at a later date.

You want to commit some code which will be merged from FreeBSD-CURRENT into the FreeBSD-STABLE branch after two weeks.

```
...

MFC after:      2 weeks
```

Where *`2`* is the number of days, weeks, or months after which an MFC is planned. The *`weeks`* option may be `day`, `days`, `week`, `weeks`, `month`, `months`, or may be left off (in which case, days will be assumed).

In some cases you may need to combine some of these.

Consider the situation where a user has submitted a PR containing code from the NetBSD project. You are looking at the PR, but it is not an area of the tree you normally work in, so you have decided to get the change reviewed by the `arch` mailing list. Since the change is complex, you opt to MFC after one month to allow adequate testing.

The extra information to include in the commit would look something like

```
PR:                 foo/54321
Submitted by:       John Smith <John.Smith@example.com>
Reviewed by:        -arch
Obtained from:      NetBSD
MFC after:          1 month
```

4. How do I access `people.FreeBSD.org` to put up personal or project information?

`people.FreeBSD.org` is the same as `freefall.FreeBSD.org`. Just create a `public_html` directory. Anything you place in that directory will automatically be visible under http://people.FreeBSD.org/.

5. Where are the mailing list archives stored?

The mailing lists are archived under `/g/mail` which will show up as `/hub/g/mail` with pwd(1). This location is accessible from any machine on the FreeBSD cluster.

6. I would like to mentor a new committer. What process do I need to follow?

See the New Account Creation Procedure (http://www.freebsd.org/internal/new-account.html) document on the internal pages.

Notes

1. The precise path depends on the `*default base` setting in your `supfile`.